CONSTRAINT CONSISTENCY TECHNIQUES FOR CONTINUOUS DOMAINS

THÈSE N^{O} 1423 (1995)

PRÉSENTÉE AU DÉPARTEMENT D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Jamila SAM

Ingénieure informaticienne diplômée EPF
originaire de Plaffeien (FR)

acceptée sur proposition du jury:

Prof. B. Faltings, directeur de thèse
Prof. P. van Beek, corapporteur
Prof. G. Gonnet, corapporteur
Prof. S. Spaccapietra, corapporteur

Lausanne, EPFL
1995

Abstract

Constraint Satisfaction Problems (CSPs) are ubiquitous in computer science. Many problems, ranging from resource allocation and scheduling to fault diagnosis and design, involve constraint satisfaction as an essential component. A CSP is given by a set of *variables* and *constraints* on small subsets of these variables. It is solved by finding assignments of *values* to the variables such that all constraints are satisfied. In its most general form, a CSP is combinatorial and complex.

In this thesis, we consider constraint satisfaction problems with variables in continuous, numerical domains. Contrary to most existing techniques, which focus on computing a single *optimal* solution, we address the problem of computing a compact representation of the space of *all* solutions that satisfy the constraints. This has the advantage that no optimization criterion has to be formulated beforehand, and that the space of possibilities can be explored systematically. In certain applications, such as diagnosis and design, these advantages are crucial.

In *consistency* techniques, the solution space is represented by *labels* assigned to individual variables or combinations of variables. When the labeling is *globally consistent*, each label contains only those values or combinations of values which appear in at least one solution. This kind of labeling is a compact, sound and complete representation of the solution space, and can be combined with other reasoning methods. In practice, computing a globally consistent labeling is too complex. This is usually tackled in two ways. One way is to enforce consistencies locally, using propagation algorithms. This prunes the search space and hence reduces the subsequent search effort. The other way is to identify simplifying properties which guarantee that global consistency can be enforced tractably using local propagation algorithms.

When constraints are represented by mathematical expressions, implementing local consistency algorithms is difficult because it requires tools for solving arbitrary systems of equations. In this thesis, we propose to approximate feasible solution regions by 2^k-trees, thus providing a means of combining constraints logically rather than numerically. This representation, commonly used in computer vision and image processing, avoids using complex mathematical tools. We propose simple and stable algorithms for computing labels of arbitrary degrees of consistency using this representation.

For binary constraints, it is known that simplifying *convexity* properties reduces the complexity of solving a CSP. These properties guarantee that local degrees of consistency are sufficient to ensure global consistency. We show how, in continuous domains, these results can be generalized to ternary and in fact arbitrary n-ary constraints. This leads to polynomial-time algorithms for computing globally consistent labels for a large class

of constraint satisfaction problems with continuous variables.

We describe and justify our representation of constraints and our consistency algorithms. We also give a complete analysis of the theoretical results we present. Finally, the developed techniques are illustrated using practical examples.

Thesis supervisor: Professor Boi V. Faltings, Artificial Intelligence Laboratory, EPFL.

Resumé

De nombreux problèmes pratiques, allant de l'allocation de ressources et à l'ordonnancement aux problèmes de conception et de diagnostique automatisés, se formulent aisément comme problèmes de satisfaction de contraintes (PSC). Un PSC se défini comme étant un ensemble de *contraintes* impliquant un certain nombre de *variables*. Le but consiste simplement à trouver un ensemble de valeurs à assigner aux variables, de sorte que l'ensemble des contraintes soient satisfaites. Dans le cas le plus général les problèmes de satisfaction de contraintes ont un aspect fortement combinatoire qui leur confère un grande complexité.

Nous nous intéressons dans le cadre de cette thèse aux problèmes de satisfaction de contraintes en domaines numériques, continus. Contrairement aux méthodes mathématiques classiques qui tentent d'identifier une solution unique et optimale, nous nous intéressons à fournir une description concise — et la plus complète possible — de l'ensemble des solutions. L'avantage d'une telle description est qu'elle ne nécessite pas de formuler de critères d'optimisation et qu'il devient possible d'explorer systématiquement l'espace des possibilités. Ces avantages présentent un intérêt fondamental pour certaines disciplines pratiques telles que la conception ou le diagnostique.

Les méthodes auxquelles nous nous intéressons pour obtenir de telles descriptions sont les méthodes dites de *"cohérence"* (consistency). Ces techniques construisent une description de l'ensemble des solutions, partielles ou complètes, en attachant à chaque variable (ou ensemble de variables) un *label* décrivant l'ensemble des valeurs admissibles. Lorsqu'un label est *globalement cohérent*, toutes les valeurs qu'il spécifie font partie d'au moins une solution du problème. Ces labels fournissent alors une description, concise et complète de l'espace des solutions et peuvent être utilisés par d'autres techniques de raisonnement.

Comme le calcul de labels globalement cohérents peut s'avérer complexe, deux approches son généralement mises en oeuvre pour en contourner la difficulté.

La première, consiste à se contenter de construire des labels *partiellement cohérent* en utilisant des algorithmes de *cohérence locale*. Ceci permet de supprimer de l'espace des solutions, des valeurs localement incohérentes. L'espace des solutions potentielles ainsi réduit peut alors être exploré à moindre coût. En domaines continus, si les contraintes sont représentées par des expressions mathématiques, l'implantation d'algorithmes de cohérence nécessite l'emploi d'outils numériques destinés à résoudre des systèmes arbitraires de contraintes. Dans le cas où les contraintes sont non-linéaires, les outils existant sont difficiles à mettre en oeuvre et posent de nombreux problèmes — notamment en matière de convergence.

Dans le cadre de cette thèse, nous nous proposons d'approximer les contraintes numériques en utilisant une décomposition hiérarchique des domaines de faisabilité — sous la forme d'arbres 2^k (2^k-trees). Cette représentation, fréquemment utilisée dans les domaines de la vision et du traitement de l'image, permet de combiner des contraintes logiquement plutôt que numériquement et évite l'emploi d'outils mathématiques sophistiqués. Nous proposons alors des algorithmes simples et stables calculant des labels de n'importe quel degré de cohérence locale.

La seconde approche consiste à identifier des classes de problèmes pour lesquelles des labels globalement cohérents peuvent être calculés efficacement. Ainsi, il est prouvé que pour des contraintes binaires, certaines formes de convexité réduisent considérablement le complexité du problème et permettent de calculer des labels *globalement* cohérents en utilisant des algorithmes de *cohérence locale*. Nous montrons dans cette thèse que de tels résultats se généralisent à des contraintes d'arités arbitraires en domaine continu. Ceci permet d'identifier des solutions polynomiales pour des classes importantes de PSCs continus.

Dans le cadre de cette dissertation, nous décrivons et justifions les représentations et algorithmes mis en oeuvre, nous effectuons une analyse théorique complète des résultats présentés et nous illustrons les techniques implantées au moyens d'exemples pratiques.

Directeur de thèse: Professeur Boi V. Faltings, Laboratoire d'Intelligence Artificielle, EPFL.

Acknowledgements

I am indebted to my supervisor, Professor Boi Faltings, who gave me the opportunity to work at the LIA and supported me throughout the years with constant guidance and encouragement.

I am grateful to Professors Peter van Beek, Gaston Gonnet and Stephano Spaccapietra, the thesis committee, for their careful reading of my dissertation and valuable feedback.

Many of the engineering issues addressed in this thesis are the result of fruitful discussions with Sylvie Boulanger from the Steel Structure laboratory (EPFL) and Dr. Ian Smith who kindly shared their experiences with me.

I would also like to thank Monika Lundell, my office mate, and my colleagues, Eric Sauthier, Berthe Choueiry, Simon Leinen, Madame Marie Decrauzat, Esther Gelle, Christian Frei, Claudio Lottaz, Ruth Stalker, Rainer Weigel, Kefeng Hua, Kun Sun, Stefan Monnier, Hidenori Yoshizumi for years of stimulating discussions, encouragement and camaraderie. A special thank to Eric Sauthier for his constructive criticism of my "Lisp programming style" and to Ruth Stalker, Monika Lundell and Christian Frei who read some "hairy" parts of the thesis, corrected my English and provided many useful suggestions.

I am thankful to my family and my friends, near and far, for standing by me.

My deepest gratitude goes to my parents for their generous, constant and unfailing support and to my husband, Salem, for his patience and comforting encouragement.

This work was supported by the Swiss National Science Foundation under contract No.5003-034269.

A Marguerite et Maurice Brügger, mes grand-parents.

Contents

List of Figures

Index of definitions

Chapter 1

Introduction

> *"Despite the illusory freedom we experience from time to time, life is, in fact pretty constrained. Luckily, in our daily doings, most of us manage to cope satisfactorily with the constraints that our wonderful world impose on us. What we need is a deeper insight into the nature of these tiny tyrants and just some more algorithms — in order to make machines see our problems in the way we do, and have them helping us." [5]*

1.1 Constraint satisfaction problems

Most decisions involved in solving the problems of our daily life are subject to restrictions. Even ordinary problems such as fixing an appointment with the dentist, placing guests around a table, or organizing a barbecue may depend upon many interdependent and potentially conflictual aspects, each of which are subject to restrictions. More importantly, each decision made to satisfy a particular aspect may impose further limitations on a problem and influence the whole resolution. When taking into consideration complex decision processes such as designing planes and space shuttles or organizing flights for an airline company, to coherently manage the multitude of underlying aspects is beyond the scope of a human's capacity and the development of efficient assistance tools is of crucial importance. This need for automated guidance tools along with the fact that restrictions, regulations and laws generally limit our decision freedom and govern our way of solving problems, has led to the emergence of popular paradigms — mainly in Operations Research, Constraint Logic Programming and Artificial Intelligence— based on the notion of *constraints*.

A *constraint satisfaction problem* (CSP) can be defined as:

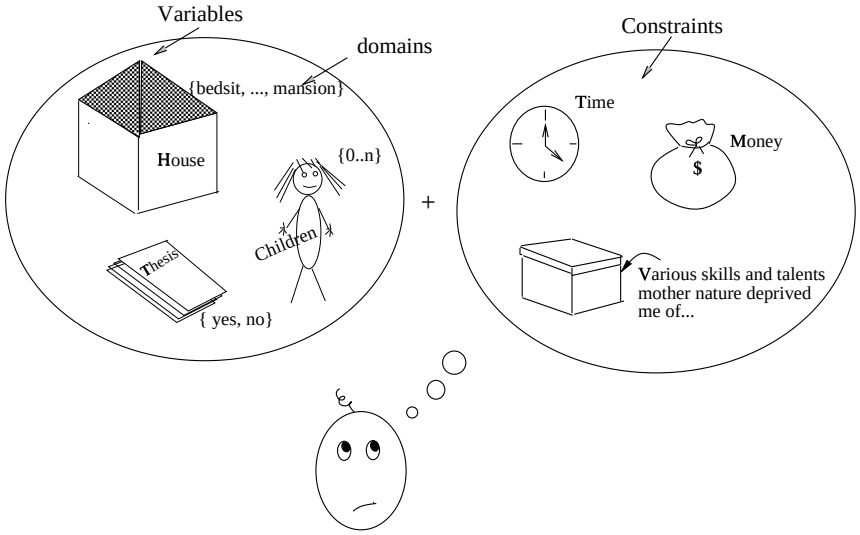


Figure 1.1: *A constraint satisfaction problem.*

- a set of n *variables* $X_1, X_2, \ldots, X_n$ taking their values respectively in n domains $D_1, D_2, \ldots, D_n$
- and a set of k constraints $C_1, C_2, \ldots, C_k$ imposing restrictions on the variables

The goal is then, to identify a set of value assignment (valuations) $(v_1, v_2, \ldots, v_n) \in D_1 \times D_2 \times \ldots \times D_n$ for the variables satisfying all the constraints (see Figure 1.1).

The constraint satisfaction paradigm allows for the natural formulation of a wide variety of practical problems, ranging from resource allocation and scheduling to fault diagnosis and design. Nowadays it is a popular paradigm in many computer science communities for its simplicity and generality.

Unfortunately, the constraint satisfaction problem includes 3-satisfiability in its most general form and is thus NP-hard. Thus, trying to solve it by systematic search algorithms for example, although theoretically feasible, often results in prohibitive computational costs.

In Artificial Intelligence one approach commonly used to overcome this complexity consists of pre-processing the initial problem using *propagation algorithms*. These algorithms establish various degrees of local consistency by pruning from the domains the locally inconsistent values. This narrows the initial feasible domains of variables, thus reducing the subsequent search effort. Using propagation techniques, one can distinguish different orders of consistency according to the number of variables checked for consistency: 1-, 2-

and k-consistency characterizing the situation where each subset of 1 (respectively 2 and k) variable(s) is consistent.

Although recent years have seen a considerable amount of activity and interest in developing efficient constraint satisfaction techniques, the results obtained so far present notable disparities with respect to the types of CSPs treated.

The vast majority of research efforts in the last years have been devoted to the case of CSPs with discrete domains which now have at their disposal relatively powerful consistency algorithms and numerous search techniques complementing each other.

In contrast, a less dynamic situation has long prevailed for continuous domains where the early attempts to apply CSP techniques have produced discouraging results. The idea of competing with the a priori overwhelming superiority of mathematical methods for solving numerical systems may have contributed to the lack of motivation for further investigations into CSP-oriented techniques. It is only very recently that new advances, notably in interval arithmetics, have motivated a renewal of activity in the field.

In this thesis, we investigate a novel approach for tackling constraint satisfaction problems with continuous variables. The next section serves as an introduction to the problem definition. It illustrates, using concrete examples, the necessary requirements underlying many practical applications and introduces the implications of these requirements on a constraint-based approach.

1.2 Example: Design as a CSP

This thesis has been motivated by shortcomings of existing techniques in a project on intelligent design systems. Design in general can be viewed as a combination of two tasks: *Conceptual design* defines the structure and its parameters, and *Detailed design* finds a set of values for the parameters in the solution (see Figure 1.2). Conceptual design manipulates partial descriptions of the artifact, consisting of parts (piers, decks, spans of a bridge etc...) and properties (construction rules, aesthetic and economic requirements, physical laws etc...).

A design task lends itself naturally to a constraint satisfaction formulation. The *variables* stand for parts and elements of the design, while the *constraints* express the properties these parts should satisfy. Solving such a *constraint satisfaction problem* (CSP)

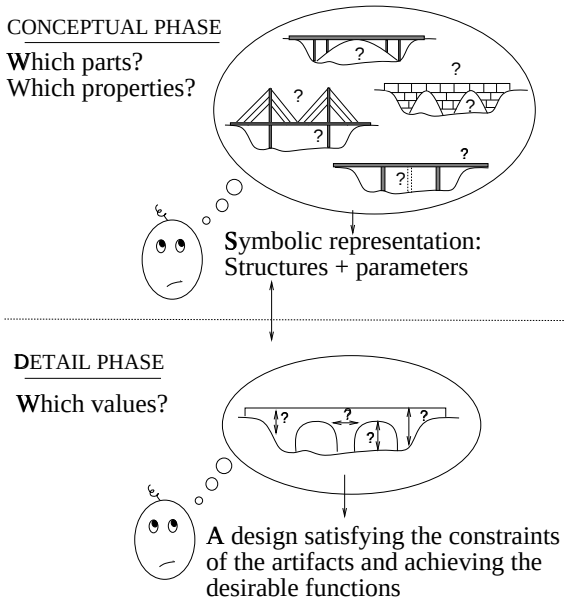


Figure 1.2: *From design to constraint satisfaction: mapping the symbolic representation used in conceptual design to actual object (detailed design) is a constraint satisfaction problem.*

amounts to mapping the symbolic representation used in conceptual design, to actual objects: solutions to the CSP determine whether and within what bounds the description produced by conceptual design corresponds to a *feasible* physical structure. In order to develop a computational model of the design process according to the constraint satisfaction paradigm, the following important factors have to be taken into account:

- The set of variables and constraints involved is not independent of particular solutions. For the design problem of Figure 1.3, going from solution a) to solution b) involves changing the number of spans and thus the set of variables and constraints. This change was made in response to variable values set in solution a) and the new variables and constraints consequently depend upon the result found while solving earlier partial problems.

- A number of variables have values in *continuous* domains (span width, deck length, pier height etc...).

- Many *inequalities* and *non-linear* constraints are involved, such as:
  $(Beam\text{-}height \geq 41.383 \left(\frac{1.1*2.35*p*Width^2}{10^6 C*f_y}\right)^{0.3976})$.

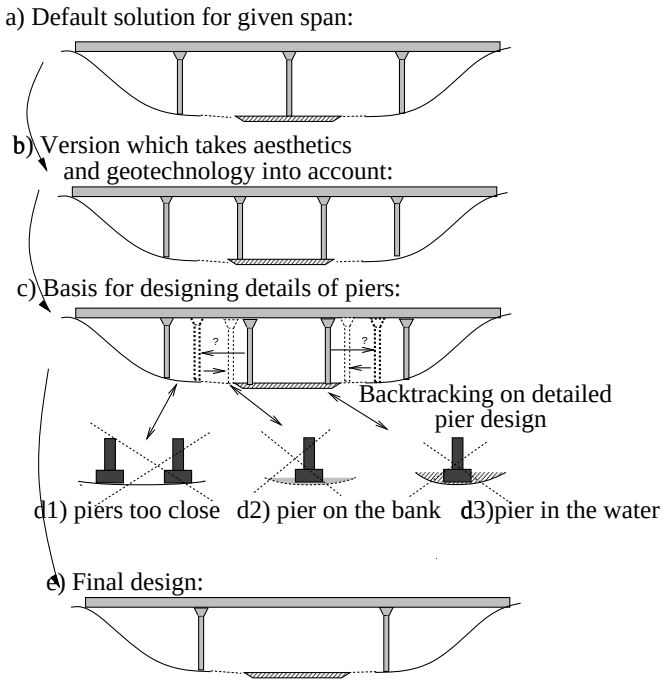


Figure 1.3: *Different phases in the design of a bridge: (i) each stage corresponds to a CSP (ii) the solution to a given CSP impacts the nature of further CSPs. The CSP of stage (c) admits no solution since designing piers for different placements is unsatisfiable.*

The first consideration shows that the CSP related to a design problem is not statically determined before the start of the solving process. Relevant information, in the form of active constraints and variables, is revealed only as the design proceeds and as decisions are taken. The *search space* is consequently given by the combination of active constraints and is generally formed by *disjoint solution sets.* Solving such a *Dynamic Constraint Satisfaction Problem* (DCSP) requires the combination of a *search process* deriving the relevant constraint combinations with *consistency checking* within each solution region.

Since numerical values are involved, the search space for DCSPs is of unbounded size a priori; each numerical value may potentially trigger a different active context. Therefore, control regimes are required to efficiently generate the relevant constraint combinations and navigate with a minimum waste of effort from one solution region to another.

In this work we do not regard these issues from a strategic perspective but rather focus our attention on the requirements they impose on the underlying partial constraint satisfaction problems. Firstly, the existence of numerical parameters creates the need for reliable constraint satisfaction techniques in continuous domains, capable of handling

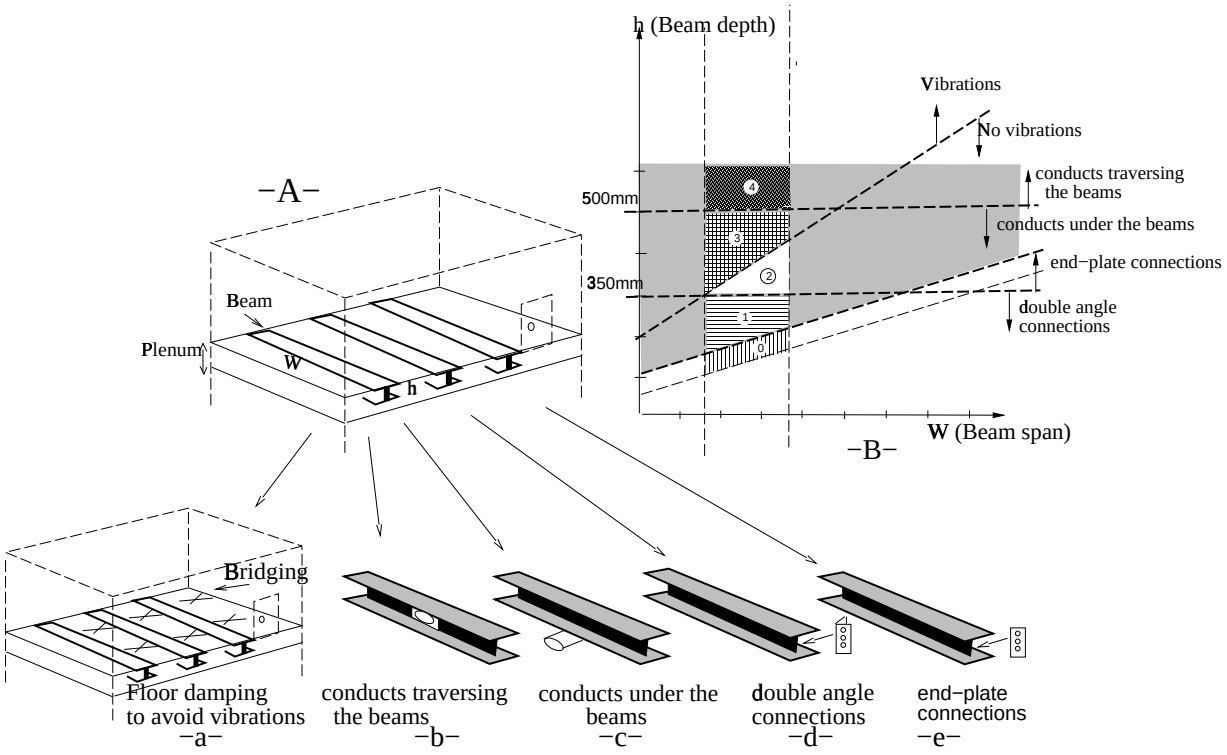


Figure 1.4: *Many continuous CSPs are embedded within complex decision processes*

arbitrary types of constraints (mainly non-linear equalities and inequalities of arbitrary arities). Secondly, since the numerical values of parameters serve as bases for decisions, the spectrum of alternatives needs to be identified as exhaustively as possible, so that the space of possibilities can be explored systematically. The example of Figure 1.4, related to civil engineering aspects of building design, illustrates this point more precisely. The structure is a floor consisting of a concrete slab on steel beams (Figure 1.4, -A-). Depending on the values for the beam depth and the beam span, different solution regions hold (Figure 1.4, -B-). Each introduces different design alternatives involving new variables and constraints:

- When floor systems become more and more slender, (i.e., the height to span ratio becomes higher (region 3 and 4)), susceptibility to vibrations increases which involves installing bridging (lateral reinforcements) for damping the floor (Figure 1.4, -a-).

- The plenum is the space situated between the floor of one story and the ceiling of the story below it (Figure 1.4, -A-). The plenum is often a fixed height, and there is a general preference for having the ventilation ducts go below the beams within this

plenum (Figure 1.4, -c-). However, for a relatively deep beam where the height (h) is greater than 500mm (region 4), it is generally no longer possible for those ducts to go under the beams. At this point, rectangular openings must be made in the beams (Figure 1.4, -b-) to allow passage of the ventilation ducts.

- There are many ways to connect one beam to another. Two factors affecting the type of connection to specify are the loading requirements and the depth of the beam. As the depth of the beam increases, a generally more economical connection, such as an end-plate[1] (Figure 1.4, -e-), becomes less feasible due to technical reasons. For instance, after a height (h) higher than 350mm (regions 2,3 and 4), a designer might consider double-angles[2] (Figure 1.4, -d-).

When numerical values strongly impact the whole solving process, the identification of single point solutions satisfying the set of active constraints is often insufficient to identify all options. In the example of Figure 1.4, a traditional mathematical method (numerical analysis, random methods) would generally identify a single point solution within regions 1, 2, 3 or 4 thus leading the designer to miss potentially better design alternatives. For such applications, it becomes of crucial importance to provide the user, or other parts of the program, with *ranges* of feasible values as sound and complete as possible. Soundness is an important requirement: in the case of our example, beam's depth and span are linked to other variables (slab thickness, beam spacing) through several non-linear equations. Considering the whole set of constraints between these variables leads to removing region 0 from the feasibility space. Partial consistency methods are not always capable of detecting this inconsistency, thus possibly leading the designer to choose a point within region 0 and take new decisions on the basis of a physically unrealizable structure.

Modeling a design task as a constraint satisfaction problem brings to light the technological requirements inherent to a wide variety of practical problems in continuous domains.

Our experience in the application of propagation techniques for civil engineering design [45, 65] along with the realization that the existing tools for *continuous* variables are unsatisfactory in several respects has motivated this research.

[1] The end-plate is a single plate which is welded perpendicular to the beam and bolted on the connecting beam or column.

[2] The double-angle connection is made of two L-shape plates which are generally welded on the beam and bolted on the column.

1.3 Problem statement

In this thesis we investigate the contribution of propagation techniques to constraint satisfaction problems in continuous domains. Variable domains are *intervals* over the reals and constraints are numerical *equalities* and *inequalities* of arbitrary types and arities.

While addressing the problem of static continuous CSP, we bear in mind that the central objective is to meet the requirements imposed by complex and dynamic decision processes, as illustrated by design tasks. In those processes, constraint satisfaction problems represent partial stages, and the solutions provided by each stage condition the further inference steps towards a global solution. Moreover, ill-formalized criteria proceeding for example from a rough statistical understanding of previous experiences, suggests *ranges* of values rather than exact single points assignments for the parameters. In practice, this translates into a large number of inequalities. Still for practical considerations, the methods developed mainly target the problems where both variables and constraints have physical interpretations and can be handled with limited degrees of precision. Note that in continuous domains *all* methods have limited precision when numerical analysis is involved.

Having stated our central objectives, we present in the following the orientations adopted with respect to general requirements. We then discuss briefly the motivation behind the choice of propagation techniques.

1.3.1 Requirements

Approximate vs exact solutions The computational floating point arithmetic precludes exactness in the mathematical sense. Except for the case of symbolic methods which produce exact solutions — their intensional descriptions — all the other methods are cast as *approximation techniques*. For linear problems, there exist methods for directly computing solutions at the floating point precision of the computer. For non-linear problems, numerical analysis provides approximations whose quality depends on the convergence of algorithms. Often in practice, rough approximations of the solutions are sufficient. To give a concrete example, the piers of a bridge are not designed at the nanometer scale and solutions given with millimetric error ranges are certainly admissible.

Completeness and soundness Our techniques are meant to address those applications where parameter values condition the decision process in which they participate. Sound approximations are consequently desirable since they would spare unintended inferences via incorrect assumptions. To guarantee soundness, variable domains must be so that each of their values participate in at least one solution. In CSP jargon, this means that the *global consistency* property must be enforced, which is shown to be NP-complete. The traditional way around this complexity barrier consists of identifying particular categories of problems that can be shifted from the NP to the P class. Research of this type have led to interesting results for discrete CSPs. In contrast, continuous CSPs have seen little effort invested in the same direction.

In approximation approaches, the tradeoff between soundness and completeness is inevitable. Choosing a conservative approximation guarantees completeness at the cost of loosing soundness (and vice-versa for non-conservative ones). More importantly the choice of one type of approximation rather than another is not always possible. By nature, the problems we address tend to be underconstrained (important proportion of inequalities) and completeness in the strict mathematical sense is not always necessary.

Generality and reliability The existing constraint solvers impose diversely sophisticated applicability conditions, at several levels. Traditional requirements generally restrict:

- the type of individual constraints (linearity, monotonicity, derivability etc...)
- the nature of the constraints systems (convexity, diagonally dominant jacobians etc...)
- the initial processing conditions.

Often in practice the restrictive properties pertaining to constraints are not, or only partially, satisfied. On the other hand, finding favourable initial conditions, which enable the solver to behave as expected, is generally a tedious task requiring a delicate handling of apparently inconsequential details. In short, choosing the appropriate solver(s) for a problem is a puzzling task, likely to confront the user with the "infamous" tradeoff between generality, reliability and efficiency [5].

The problems we are interested in do not limit the constraints to particular classes and involve a large number of non-linearities. This explains why most of the solvers we can resort to are unsatisfactory with respect to our general requirements. As for technical matters, a general objective in this thesis was consequently to provide the capability of handling arbitrary types of constraints with emphasis put on computational reliability. This led us to prospect for properties of the feasibility space, rather than analytical properties of constraints, making these objectives attainable in reasonable time.

Scalability Often the scalability of a constraint solving method is discussed in terms of the number of variables or constraints involved (leading some early authors to evoke "the curse of dimensionality" [6] for instance). NP-completeness implies inevitably explosive combinatorics in the worst cases. The "real" problems are however more likely related to the nature of the solution spaces rather than to size considerations. Large problems with high solution rates might be easier to solve than smaller tight ones for example. In this thesis we will devise these scalability issues in terms of topological properties of the solution space.

1.3.2 On the advantages of consistency techniques

A constraint satisfaction problem can have one, several, or no solutions. When there are several solutions — as in the case of underconstrained systems — a solution method might return only one solution, possibly optimal according to some criterion. Identifying single point solutions is the viewpoint adopted by almost all the existing *mathematical solvers*, ranging from linear and non-linear programming methods to numerical analysis and stochastic techniques.

As discussed before, we focus on the problems where optimization criteria are difficult to formulate beforehand and where the spectrum of possibilities need to be recognized as precisely as possible. These requirements rule out the afore mentioned mathematical tools as appropriate technologies.

In contrast to the traditional mathematical methods, constraint satisfaction techniques in continuous domains aim at constructing a compact description of the space of all solutions. This is done by assigning one or a small collection of feasible intervals to the variables, thus making them attractive for the types of problems we address.

However, while extensively applied to discrete problems, these techniques have seen marginal research interest in the direction of continuous constraints. This is not surprising when we consider the fact that constraint propagation algorithms were proposed originally as preprocessing procedures for enabling tractable search in enumerable spaces. Although continuous feasibility spaces preclude enumeration in the strict sense, the notion of constraint propagation for *local pruning* remains appealing also for continuous problems, as rapid and flexible approximation techniques.

Flexibility is one of the main arguments for the choice of consistency techniques rather than the alternative interval-based techniques proposed by *interval arithmetic*. In effect, consistency techniques construct *partial, locally consistent solutions*, recurrently extensible. This brings them the possibility of tuning the quality of the results as a function of the application complexity. As opposed to this approach, interval arithmetic techniques handle the entire problem as an indivisible whole and scalability problems become unavoidable. Moreover, a particular care must be exercised while using interval-arithmetic techniques to prevent unrealistic estimates of the solution space.

Of course this does not mean that traditional techniques are not useful and we believe that the appropriate combination of complementary technologies is the key to the emergence of powerful solvers. The alternative schemes mentioned (linear and non-linear programming, interval arithmetic etc...) and countless hybrid combinations have been used successfully in many applications. However as the types of problems described above are attacked, extending the potential contribution of consistency techniques to the issue seems to be of great interest.

These issues are discussed more extensively in chapter 2.

1.4 Contribution

Recent advances have begun to reconcile CSP techniques with non-discrete domains. Yet, due to the difficulty of dealing with arbitrary numerical constraints, little effort has been invested into the direction of:

- designing propagation algorithms for higher degrees of consistency than 2-consistency[3],

[3]2-consistency takes into account pairwise compatibilities between variables.

- identifying conditions — and how they can be verified — under which global consistency is tractable.

The central contribution of this thesis is the proposal of a framework enabling the transference of techniques and results from discrete to continuous problems. Algorithms for 3- and higher degrees of consistency in continuous domains are then proposed within this framework.

In continuous domains, reaching global consistency is especially desirable as traditional search techniques cannot be used to enforce subsequent refinements of the results. The theoretical side of this work is devoted to identifying a general class of continuous CSPs for which this property can be reached tractably. A decomposition schema is also proposed for solving the problems falling outside this class. Previous works on discrete and temporal continuous problems have inspired these results.

Below is an outline of the main areas that this thesis investigates.

1. *Conditions for tractable global consistency.* In *consistency* techniques, the solution space is represented compactly by *labels* assigned to individual variables or combinations of variables. When the labeling is *globally consistent*, each label contains only those values or combinations of values which appear in at least one solution. Such a labeling is a compact, sound and complete representation of the solution space admitted by the constraint satisfaction problem, and can be combined with other reasoning methods. In practice, computing a globally consistent labeling is too complex, and the labels may include values which do not appear in any solution. Recent studies on discrete and temporal continuous CSPs have proved the value of convexity properties for enforcing global consistency in polynomial time using low orders of consistency. We propose an extension of these results to the case of general continuous CSPs.

2. *Constraint representation.* In discrete domains, labels are represented simply as enumerations of values or value combinations. In continuous domains, sets of individual values are often compact and can be represented by one or a small collection of intervals. However, representing and manipulating labels of several variables is more involved as they may be complex geometric shapes. Standard interval-based methods roughly approximate a constraint by an enclosing box whose sides correspond

to the unary outer projections of each variable involved. This representation does not allow the proper implementation of algorithms for more than 2-consistency. An alternative representation, inspired by standard methods in image processing and vision, is proposed that overcomes these limitations. This approach, in contrast to the commonly used ones, maintains a clear distinction between the feasibility domains of variables (unary projections) and the effective solution space. It enables an easy combination of simultaneous constraints using logical, rather than numerical, operators.

3. *Consistency techniques.* In discrete domains, while the results on convexity showed great potential for binary constraints, generalization to constraints of arbitrary arities has posed complexity problems. We show that in continuous domains, it is possible to remedy this state of affairs using a different generalization approach. A particular form of consistency, called $(r, r-1)$-relational-consistency is discussed. It enforces global consistency in polynomial time under some identified convexity conditions.

4. *Decomposition schemes for general continuous CSPs.* For problems satisfying no convexity properties, the worst case theoretical complexity remains exponential in general, which means that time complexity is hard to predict for particular instances. We sketch out a simple method to decompose general problems into sub-problems with interesting convexity properties. From a practical perspective, this decomposition provide a means to *i)* reduce the complexity by applying search technique along with their numerous enhanced schemes, *ii)* predict *a priori* the resolution complexity.

1.5 Guide to the thesis

This section presents the content of each of the remaining chapters:

Chapter 2 surveys the techniques from several different fields contributing to the issue of constraint checking in continuous domains. In this chapter emphasis is put on constraint satisfaction techniques, but the state of the art from other areas is also devised more briefly in order to set a comparative framework. The methods mentioned target the same generic

problem, even if they have radically different philosophies, and play a practical role at the present, especially within the context of constraint programming languages.

Chapter 3 discusses the problems inherent to the constraint representation traditionally used in consistency techniques with interval labels. It proposes an alternative representation based on a hierarchical decomposition of the solution space into 2^k-trees. This chapter also shows how the 2^k-tree representation of constraints can be constructed automatically.

Chapter 4 devises several consistency algorithms based on the 2k-tree representation of constraints. Algorithms for 2-, 3- and k-consistency are proposed, discussed and compared with their existing counterparts.

Chapter 5 shows how global consistency can be reached using low order consistency algorithms, provided that convexity properties are verified. First stated for binary constraints, these results are shown to admit polynomial time extensions to n-ary constraints, contrary to the case of discrete domains. A general classification of continuous CSP, in terms of resolution complexity, is then presented. This chapter also proposes a decomposition scheme for solving problems verifying no simplifying convexity properties.

Chapter 6 describes the implemented system and reports the application of our approach to practical problems borrowed from the field of mechanical and civil engineering design.

Chapter 7 reviews the contribution of this thesis, discusses some open problems and suggests some future issues to be studied.

Chapter 2

A Survey of Constraint Solving Techniques

2.1 Introduction

Resolution techniques for numerical CSPs encompass a large number of different areas including numerical analysis, linear and non-linear programming, interval arithmetic, evolutionary programming, operations research and consistency algorithms. Although our work contributes mainly to the latter issue, we start the review of related works by sketching out briefly the prominent techniques from other disciplines, in order to set a comparative framework. The works reviewed have been chosen for their practical contribution to operational solvers and constraint based languages. This chapter will then present more exhaustively the techniques and literature directly related to our work.

2.2 Conventional mathematical techniques

The survey of methods for solving numerical systems starts with a general review of the classical mathematical methods. We gather under this denomination: linear and non-linear programming methods as well as algebraic and iterative numerical methods. Linear systems have at their disposal efficient tools among which Danzig's simplex algorithm [31] (linear programming) and Gaussian elimination (linear algebra) play a central role. Linear programming methods [125] provide optimal solutions for a set of linear equations and inequalities with respect to an objective function. Linear algebra embodies a systematic solution method — based on iterative syntactic replacements — that works for any solvable linear problem (Gaussian elimination for equalities, Fourier-Motzkin [125] for inequalities

along with numerous variants). These methods are implemented in a wide range of mathematical packages (Macsyma [43], Mathematica [94], Maple [17], Reduce [66], Gams [12] ...) and in numerous Constraint Logic Programming (CLP) languages.

Unfortunately, non-linear systems offer far more resistance to conventional mathematical treatments: algebraic methods are restricted to polynomials, numerical methods offer no convergence guarantees and may exhibit chaotic behaviors in general. Finally, reliable non-linear programming methods (descent and direct methods [4, 53, 89]) are restricted to convex solution spaces.

These limitations do not mean that conventional techniques are not useful, but they have been considered sufficiently dissuasive by certain practical systems (mainly CLP languages) which simply choose to radically avoid the problem, either by:

- restricting themselves to linear constraints (CLP(R) [76], CHIP [42, 140]),
- delaying the evaluation of non-linear terms in the hope that they become linear after a number of variables have been properly grounded (by solving separately the linear terms) (PrologIII [25, 26]),
- or taking advantage from the possibility of constructing easily enclosing linear approximation for certain types of constraints (quadrics) (QUAD-CLP(R) [109]).

Nevertheless, as we will see in the following sections, many other systems attempt to provide some support to non-linearity by including either numerical techniques or some recent algebraic techniques.

2.2.1 Algebraic algorithms for non-linear constraints

The last three decades have seen the development of two algebraic methods for solving non-linear constraints over real numbers: the method of Gröbner bases for equalities and the method of partial cylindrical decomposition for inequalities. These methods are devoted to polynomial constraints.

In [72], Hong reports the integration of these two methods in the constraint logic programming language Risc-CLP(Real) [71].

Gröbner bases have been introduced by Buchberger [15, 14] to transform a set of polynomials into a certain standard form with better computational properties. This concept

facilitates notably the detection of unsolvability in systems of equations. In [14], Buchberger proposes a practical algorithm for computing Gröbner bases which generalizes the notion of Gaussian elimination for linear multivariate equations. This work has initiated several works leading to improvements and generalizations [29, 54, 32]. Buchberger's algorithm (or its variants) is integrated in mathematical packages such as REDUCE [95] and constitutes a core of the CLP language CAL [117]. In [123], Sato and Aiba report the successful use of CAL for solving complex kinematic problems in tens of seconds.

Solving a system of non-linear *inequalities* is formally defined as a quantifier elimination problem in the first order theory of real closed fields for which Tarski [133] proposed the first algorithm. Although this work was of no practical use by reason of an intractable inherent complexity, it initiated an abundant literature and numerous further researches among which the algorithms proposed by Collins [24], Grigorev *et al.* [55, 56] and Renegar [113] represent the most significant advances. Inspite of its double exponentiality, the CAD (Cylindrical Algebraic Decomposition) algorithm, due to Collins, is the best among these algorithms and was the first to be practically and theoretically useful. It has given rise to numerous improvement [70, 84] or partial variants and enhancements [85, 23] providing asymptotically fast algorithms for certain subtasks. Note finally that the variant of the CAD algorithm proposed by Hong [70] and called Partial CAD (PCAD) has led to dramatic algorithmic improvements — obtaining speed up to 300'000 times for certain complex problems [72]. PCAD is the core solver of Risc-CLP(Real) [71] which also includes Buchberger's algorithm for non-linear equalities.

A more extensive survey of the state of the art algebraic algorithms for checking non-linear systems of equalities and inequalities is given in [69].

Discussion Although algebraic techniques offer the major advantage of compiling solutions for certain topologies of constraints, thus offering a run time significantly faster than iterative techniques when used repeatedly, they are plagued by a poor scalability and restrictive applicability conditions (polynomials etc.). In effect, for arbitrary constraints, computer algebra is a computationally complex domain: it is shown that the canonical simplification of expressions involving transcendental functions is undecidable and that the factorization of multivariate polynomials has a worst case exponential complexity [16]. This explains why in practice, algebraic techniques tend to be used more as

solution checkers than as solution generators [80].

2.2.2 Iterative numerical algorithms

The most compelling advantage of iterative numerical techniques is their generality: in effect, they are theoretically applicable to any problem which can be represented as a set of equations. Numerical iterative techniques deliver pointwise approximations calculated throughout series of refinements converging, in the best case, towards the exact solution. Since these methods target mainly the issue of solving systems of *equations* and produce pointwise approximations, we are more interested in their interval arithmetic counterparts presented later. This section only gives a succinct and general overview of the major techniques used in practice.

The vast majority of iterative techniques are devoted to unconstrained minimization [77, 105], of which the simultaneous non-linear equations problem is a particular instance. Less is known about how the constrained version, where each variable is subject to restriction, should be solved. Numerical iterative techniques typically require methods to evaluate the problem functions along with a starting point considered as a crude approximation to the solution. These techniques are cast either as [77]:

- Global methods: used when the starting guess is not close to the solution and which guarantee convergence to a local minimizer of a non-linear functional or some solution of a system of non-linear equations from almost any starting point (bisection).
- Local methods: which can be used only in the vicinity of the answer (Newton, Broyden, Gauss-Jacobi, Gauss-Seidel [50, 129, 108].

It is acknowledged that purely global methods are in general too inefficient for general use [3] and the issue of how combining effectively local and global methods is central for managing properly the tradeoff between efficiency and reliability (quasi-Newton, quasi-Newton using Broyden's update, etc. [40, 77, 13]).

Discussion The most significant shortcomings of iterative numerical methods can be summarized as follows:

- The solutions found are correct within a convergence tolerance and the non-existence of a solution can only be inferred through a failure to converge after a fixed maximum

number of iterations.

- Predictability in terms of convergence and convergence speed is hard to establish since the number of iteration required to solve a problem can vary substantially depending on the initial conditions given to the solver.
- Iterative solvers cannot distinguish multiple solutions which sometimes lead them to jump from one solution to another and to initiate a chaotic behavior.
- Some iterative solvers require calculating partial derivatives generally obtained through finite difference methods.

Inspite of their disadvantages, these methods are among the most widely used for dealing with non-linear engineering problems because of their generality. Numerical techniques have been extensively studied for decades and many researchers believe that "the field has gelled to a point where [...], while some improvement is still likely, one no longer expects new algorithms to result in quantum jumps over the best being used today" [77]. It appears however that these techniques now offer valuable guidelines for improving both interval arithmetic and constraint satisfaction techniques, as we will see in the following sections.

2.3 Stochastic methods

Stochastic methods refer to a class of generally incomplete and non-systematic search techniques including heuristics and non-determinism. Examples of such techniques are hill climbing, tabu search, simulated annealing and genetic algorithms. These techniques can be considered as adaptive in the sense that they start with a random focal point in the search space and modify it repeatedly using heuristics until it corresponds to a solution (within a certain maximum number of iterations). These methods are generally robust and good at finding a global minimum in large and complex search spaces. In [134], Thornton reports the successful use of genetic algorithms and simulated annealing techniques for solving large systems of non-linear inequalities derived from mechanical design problems. As the name *stochastic* implies, these methods have a random aspect which reduces the chance of converging to local minima. However, situations may occur where the search process is in a wrong portion of the solution space. This generally requires the system

to be restarted from randomly selected starting points. The non-systematic search of pointwise solutions does not allow the construction of a compact description of the set of feasible alternatives in the solution space. Stochastic methods are consequently out of the scope of this study.

2.4 Interval arithmetic

The first practical applications of interval arithmetic date back to Moore [100, 101]. While the original purpose of this area was to quantify the errors introduced by finite precision arithmetic, it also appeared to constitute an appropriate framework for dealing with uncertainty [102]. Interval arithmetic techniques address a wide variety of problems such as linear and non-linear interval equations [2, 106, 10], inclusion of the ranges of functions and their derivatives [101, 111] as well as optimization [112, 51].

In interval arithmetic, real numbers are approximated by intervals and arithmetic expressions are computed by applying the operators of the formula to the endpoints of the intervals of its arguments — for example, [2..4] - [1 2] yields [0..3].

Most properties of arithmetic operators (commutativity and associativity) carry over to their interval counterparts. However, distributivity does not hold in general and is replaced by a weaker property called sub-distributivity ($A(B + C) \subseteq AB + AC$). This explains why naive evaluations of interval arithmetic expressions often result in strong overestimates. Since for example $A(1 - A) \subseteq A - A^2$ [10], the determination of the operations ordering for obtaining the best computable enclosure can be a quite complex task for arbitrary expressions.

2.4.1 Solving linear interval equations

Systems of linear algebraic interval equations can be solved either by direct inversion or using the interval counterparts of the standard iterative algebraic techniques.

The former consists merely of computing the direct inverse solution using interval matrix inversion. A variant of direct inversion is also given by the Krawczyk's operator [81, 82] which often yields sharper results when applied iteratively. These simple methods require the regularity of the matrix to be inverted — a condition which does not hold in general — and are consequently too restrictive for general use [106].

A more general alternative consists of applying the extension of Gauss-Seidel [114] or Gauss-Jordan [62] iterations to interval systems — and eventually replace the iterative solution process by direct solutions when regularity is detected.

Systems of linear interval equations can also be solved using the interval extension of Gaussian elimination [106], but the algorithm requires preconditioning and can be carried out reliably only when the matrix is strongly regular[1] . When these conditions hold, Gaussian elimination remains the preferred solution.

2.4.2 Solving non-linear interval equations

Most of the resolution methods for systems of non-linear equations are direct extensions of the standard numerical iterative techniques, succinctly presented in section 2.2.2. Interval arithmetic counterparts exist for many locally convergent numerical methods such as: Newton [1], non-linear version of Gauss-Seidel (Hansen-Sengupta operator [61]), Gauss-Newton [1]. The Krawczyck's non-linear operator is often used to solve a step of the Newton interval iterate. Combined with interval Newton, this operator has indeed proven to be useful for producing good evaluations of complex interval functions. Most of the state of the art interval arithmetic techniques are direct successors of the Krawczyck's method [61, 60].

Bisection [78] remains the main globally convergent technique used in interval arithmetic. It consists of recursively bisecting the original domain in the direction corresponding to the largest interval, while trying to remove the sub-domains having no intersection with the interval enclosure of the effective solution space. Similarly to the case of globally convergent numerical techniques, the slow convergence of bisection precludes its utilization alone for solving practical problems. This method often complements locally convergent iterate, notably to identify multiple solutions.

Discussion Interval arithmetic methods compile compact enclosing approximations of the solution space. They are consequently appealing when it is needed to reason on a spectrum of possible feasible alternatives rather than on single point isolated solutions. These methods tend to prefer completeness over soundness since the interval enclosing representation of the solution space loses pointwise relations between variables' values.

[1]Let $\mathcal{M}$ be an interval matrix and $\mathcal{M}_c$ be its centered form. $\mathcal{M}$ is said to be strongly regular when the matrix $\mathcal{M}_c\mathcal{M}$ is itself regular (strict diagonal dominance) [10]

From the efficiency and robustness standpoints, interval arithmetic techniques present roughly the same advantages and shortcomings as iterative numerical techniques —from which they mainly derive. Their major disadvantage remains that special care must be exercised in advanced applications to prevent unrealistic estimates:

- When a constraint expression contains multiple occurrences of the same variable, simple interval evaluation often result in excessively loose bounds: a simple expression such as $X - X$ can generally not be recognized as zero and the global minimum of -1/2 for the expression $X^2 - X$ will not be detected [116, 128].

- In general there is no guarantee that the sequence of intervals generated by locally convergent operators (Krawzcyk, Hansen-Sengupta, etc.) will converge at all. Hence, the determination of appropriate initial domains is of crucial importance for non-linear systems [103]. This problem — generally addressed using bisection — is analog to the choice of proper and safe starting conditions for iterative numerical techniques.

Finally, both linear and non-linear algebraic interval methods require preconditioning steps or may impose sophisticated applicability conditions (regular matrices, globally dominant Jacobians).

2.5 Consistency techniques

We now tackle the issue of consistency techniques to which this work is more specifically dedicated.

While the approaches previously presented handle the entire set of constraints as a whole, hence providing global solutions, consistency techniques are based on the construction of *partial solutions*, recurrently extensible, which brings them far more flexibility.

In effect, the advantages in terms of accuracy and soundness provided by global solution constructors are often mitigated in practice by the fact that:

- they require rigid representations difficult to handle by non-experts,

- they pose scalability problems and,

- they often involve opaque and complex processes where the user has no control over the decision mechanisms.

Constraint based approaches in AI can be traced back to the development of the first interactive graphical interfaces using geometrical constraints (Sketchpad [131]) in 1963. Later several computer-aided design systems used the notion of constraints as problem specifications. The first formalizations of constraint satisfaction problems and techniques were issued from research on line labeling in vision [73, 22, 142] which also originated a basic consistency technique called the Waltz filtering algorithm (also referred to as AC-2). This research on image processing led Montanari [99] to propose the first systematic treatment of constraint networks along with a path-consistency algorithm called PC-1. This work initiated an abundant literature and gave strong impetus to research on constraint satisfaction problems.

In the following we outline the main issues dealing with constraint satisfaction problems. For a deeper insight to the subject, we refer the reader to [91, 63, 64, 83, 96, 135].

2.5.1 Basic terminology

Representation Constraint satisfaction problems are generally represented using *graphs* or *hyper-graphs*, depending whether the constraints are binary (affect pairs of variables) or n-ary (apply to n-tuples of variables). These graphs are such that the nodes represent the variables involved in the problem while the arcs or hyper-arcs stand for constraints between variables. The vast majority of research on CSPs is devoted to *binary* problems and less is known about how n-ary problems should be solved. Non-binary discrete problems can be mapped to binary ones — with generally more complex variable domains — using reduction methods [115]. On the other hand, the simple transformation method based on projection[2] applies to general problems but corrupts the effective solution space. A *static* CSP is a problem where both sets of variables and constraints are entirely pre-determined and do not vary during the resolution process. In contrast, a dynamic CSP [97] may see some constraints added or removed during the problem solving process while certain CSPs may be subject to dynamic constraints [57] — which are constraints that are themselves subject to constraints. Although dynamic CSPs fall out of

[2] An n-ary constraint can be replaced by its set of binary projections. When re-composed, these projections generally yield a region enclosing the original n-ary constraint.

the scope of this thesis, a previous experience in applying this concept to practical design problem has initiated our investigations on continuous CSPs.

Domains and labels In a CSP, a distinction is made between the *domain* of a variable and its *label*. The first notion refers to the whole set of values the variable can take, while the latter represents the subset of the domain assigned to the variable at a given time. Both labels and domains can be either discrete or continuous (subset of $\Re$).

Solutions We consider a CSP,(V,C,D), where the triplet (V,C,D) denotes the sets of variables, constraints and domains respectively and where $V = \{x_1 \dots x_n\}$. We denote as L_i the label of a variable x_i. A globally consistent solution to the CSP is given by an instantiation $\{v_1 \dots v_n\} \in L_1 \times \dots \times L_n$ of the variables satisfying all the constraints of C. A partial instantiation of k variables $\{x_1, \dots x_k\}$ is also called a k-instantiation.

It represents a partial solution if it is consistent with all the constraints involving $\{x_1, \dots x_k\}$.

2.5.2 General consistency techniques

Depending on the objectives fixed by the application, a CSP may be either an *enumerative* problem where the goal is to identify the whole set of globally consistent solutions, an *exemplification* problem if a limited number of globally consistent solutions is sought or a *decision* problem if the task consists of verifying that a given instantiation is globally consistent [11]. Many engineering problems require the space of possible alternatives to be explored systematically and are thus enumerative constraint satisfaction problems (see chapter 1). Exemplification and decision numerical CSPs can use mathematical tools such as linear programming which return pointwise solutions, eventually according to certain optimization criteria, or algebraic checking algorithms. When enumerative problems are taken into consideration, tools for constructing a compact description of the effective feasibility space become desirable and interval-based CSP techniques seem to be more appropriate. In the following, we start by presenting general constraint satisfaction techniques. A survey of the state of the art methods in continuous domains is then given.

Problem reduction techniques

Backtrack search, along with its numerous variants and enhancements, remains the principal mechanism for solving a CSP. However, since it is often the source of intractable complexities, an appealing idea called *problem reduction* has been introduced with the purpose of transforming the initial problem into a hopefully easier equivalent one (or easily recognizable as insoluble one) [135]. Problem reduction consists merely of pruning from the domains of variables the values that are locally inconsistent, thus reducing the solution space as well as the subsequent search effort. These methods exhibit polynomial time complexities — where the polynomial degree is a linear function of the level of local consistency checked for. This notion of problem reduction came in the literature along with formal definitions of local degrees of consistency in constraint networks and local consistency checking algorithms. Several degrees of local consistency have been defined: *node-*, *arc-* and *path-* consistency defined by Mackworth [90] and generalized to *k-consistency* and *strong k-consistency* by Freuder [46]:

Definition 2.1 (Freuder) *A network is k-consistent if and only if, for any given instantiation of any $k-1$ variables satisfying all the direct relations among those variables, there exists an instantiation of the k^{th} variable so that the k values taken together satisfy all the relations among the k variables.*

Definition 2.2 (Freuder) *A network is strongly i-consistent if it is i-consistent for all $i \leq k$*

Strong 2-consistency and strong 3-consistency correspond respectively to arc and path-consistency.

From the algorithmic point of view, arc-consistency techniques have been the subject of particular focus and attention. Since the first arc-consistency algorithm by Waltz [142], numerous variants have been proposed: AC-1, AC-2, AC-3 described in [90] and [92], further improved in the discrete domain to AC-4 [98], generalized to AC-5 [41] and culminating with AC-6 [9].

The first path consistency algorithm PC-1 — further refined to PC-2 [90, 92] — has been proposed by Montanari in [99]. In the discrete domain, PC-2 is itself improved to PC-3 [98] and since it turned out that slight errors have been made in this algorithm, a corrected version is presented in [59].

Algorithms for higher degrees than arc consistency require tools for synthesizing constraints (i.e., inferring constraints of higher arities than the ones originally present in the network, by composition). This notion has been introduced by Freuder in [46] who proposed the first synthesis algorithm, refined later by Cooper [27]. The process of inferring the constraint of arity n (n-constraint) for a problem of n variables, by extending partial solutions, is also referred to as solution synthesis [135]. Solution synthesis is naturally space and time demanding in general and while using problem reduction techniques, an acceptable compromise between the effort devoted to local pruning and the cost of subsequent searches must be sought. In practice, the most commonly used algorithms are those enforcing arc- and path-consistency.

Simplifying properties and backtrack-free search

Extracting a particular solution from a consistent labeling is an iteration of two steps in which values are assigned to variables sequentially. In the first step, an unassigned variable is selected and assigned a value within its label. In the second step, the labels of all remaining unassigned variables are updated so that they contain only values which are consistent with those already assigned.

If the initial labeling is globally consistent and nonempty, every partial assignment of variables can be extended to a full solution. Consequently, the assignment procedure will never require any backtracking. Computing globally consistent labeling amounts to synthesizing the n-constraint for an n variable problem, a task which exhibits an exponential time complexity in the most general case.

However, provided that certain conditions are verified, local consistency algorithms are sufficient either to compute a globally consistent solution space or to bound the remaining search effort.

For example, it has been shown [47] that when the constraint network is a tree, arc-consistency also guarantees global consistency. This is an example of how the topology of the constraint *network* can simplify the computation of a globally consistent labeling [46]. Similarly, path-consistency is sufficient to ensure backtrack-free search in series-parallel networks. It has also been shown that when the constraints themselves are convex and binary [137, 38] path-consistency ensures global consistency irrespective of the topology of the network.

These simplifying properties deal with the type of constraints (monotone, functional, convex, etc. [99, 36, 38, 28, 137, 139, 141, 41]), topological characteristics of the constraint graph [47, 48], or a combination of both notions. Chapter 5 presents some results falling under the class of works identifying such properties. It contains a more extensive review of the subject.

The complexity of consistency algorithms may also be reduced using domain specific characteristics. As established by Dechter and Pearl [37], ensuring node and arc-consistency is stronger than required for enabling backtrack free search in tree-structured networks. They consequently propose a weaker concept called *directional arc-consistency* which guarantees backtrack-free search in trees. This notion generalizes to higher degrees of consistency and is defined under total ordering of the variables.

Other preprocessing techniques

In addition to problem reduction techniques, the complexity of constraint satisfaction problems can be reduced using several methods including:

- *graph-reduction techniques:* the constraint graph is simplified prior to search using reduction techniques such as vertex elimination or vertex merging (e.g., [18]),
- *abstraction techniques:* the size of the search space is reduced by clustering values, variables or both (e.g., [20]),
- *decomposition methods:* topological properties of the constraint graphs are used to propose decomposition schemes able to reduce the complexity of the problems (e.g., [20, 49]).

These methods are out of the scope of this thesis and will not be described in more detail here.

2.5.3 Consistency techniques in continuous domains

As mentioned in the introduction (chapter 1), the early attempts to establish constraint propagation as appropriate technology for continuous domains have shown little success: traditional consistency techniques and propagation algorithms — such as the Waltz propagation algorithm — provide relatively poor results when applied to continuous domains:

they ensure convergence and completeness for only restricted classes of constraints (e.g., bounded differences [34]). When non-linear and transcendental constraints are involved, the behavior of these algorithms becomes unpredictable and is often plagued by the two following problems:

- *early quiescence*: the refinement process stops prematurely and no consistency is ensured,
- *cycling*: the refinement process enters infinite loops

A good insight into the problems encountered can be found in the paper of Davis [34].

For a long time these negative results have been attributed to the analytical complexity inherent to solving general systems of equations and inequations. Recent advances [75, 44, 88, 7] have however mitigated these conclusions and proposed various remedies and significant improvements.

Yet, since composing continuous constraints numerically is a complex task, most of the techniques for continuous CSPs work with the unary projection of constraints rather than constraints themselves and the vast majority of propagation techniques are variations of the same theme — fixed-point iteration via arc-consistency algorithms — with emphasis put on improving the tightening of the outer domains bound for variables. Constraints are generally roughly approximated by the feasibility domains of the variables they involve and the combination of constraints then reduces to elementary operations on intervals.

2.5.4 Arc-consistency in continuous domains

The improvements of arc-consistency techniques mentioned in section 2.5.2 all exploit features inherent to discrete solution spaces, by the introduction of explicit counters and supports for the label's values for example. They are consequently not directly applicable to the case of continuous CSPs. This explains, from an algorithmic standpoint, why propagation algorithms have not evolved in continuous domains and still take their general original forms (mainly AC-3). The research efforts have in general focused on the issue of enhancing the constraint revision operators used during the relaxation steps of consistency algorithms.

The first practical implementations limited themselves to using basic interval arithmetic operations for refining the intervals labels, through fixed point iterates

(BNR(prolog) [107] and its successor CLP(BNR) [8], CLP(R) [87]). Since elementary interval arithmetic operations often produce unrealistic estimates, these implementations are relatively weak. They fail to narrow the labels for this kind of simple linear problems [19]:

$$\begin{aligned} X + Y &= 5 \\ X - Y &= 6 \\ X, Y \in (-\infty, \infty) \end{aligned}$$

and cannot detect the following types of inconsistencies [19]:

$$\begin{aligned} A + 1 &= D \\ A + B &= D \\ A \in (0, \infty) \\ B \in (-\infty, 0) \end{aligned}$$

They also assume a single, unsplittable label (interval) per variable.
Variants of these systems have been implemented based on:

- A specific adaptation to particular types of constraints like in CLP(F) [68] which is devoted to functional constraints and which ensure a better narrowing, in some cases, by putting interval constraints on the values of the constraint derivatives at points and intervals.
- A capability of handling multiple solutions like in ECHIDNA [127] where real domains are represented by interval hierarchies allowing control of the constraint processing precision and providing a hierarchical version of consistency algorithms. Here again, the narrowing operators are based on elementary interval arithmetic and the results provided for the two simple examples mentioned before are not better.

Numerical search

A first remedy to the absence of narrowing is proposed by Cleary [21] who uses a case analysis technique based on domain splitting. Cleary considers relational arithmetic on

continuous domains by adapting techniques from interval arithmetic. These techniques are introduced into Prolog for narrowing the intervals associated with constraints. Narrowing is done using specific algorithms adapted to each kind of constraint and these techniques are coordinated with a backtracking tree search: intervals for constraints are split into two, and one interval is visited after the other upon search. Cleary's algorithms have been generalized for narrowing intervals constrained by any relation on $I(\Re)^n$ [87, 86] and incorporated in the constraint logic programming languages CHIP and CLP($\Re$).

Hybrid techniques

Since domain splitting and its subsequent backtracking tree search is costly certain constraint logic programming languages limit the use of propagation techniques to non-linear constraints and use powerful mathematical tools for linear constraints (CIAL [19]). CIAL includes a linear solver based on preconditioned Gauss-Seidel method in addition to an interval narrowing module devoted to non-linear constraints. When some variables become non-narrowable, they are considered as constant terms — which might render linear some non-linear constraints and activate again the linear solver. Such hybrid implementations provided significant improvements for certain types of problems [19].

Some authors have also attempted to improve the results of interval-based consistency techniques by combining them with assortments of numerical tools such as relational and constant elimination arithmetic [128, 124], symbolic algebra [30, 122]), hierarchical strategies for handling inequalities [116] or domain-specific interval propagation tactics [143].

The state of the art

The most significant improvements of arc-consistency techniques in continuous domains are given by the works of Lhomme [88], Benhamou *et al.* [7] and Faltings [44]. The two former works present several analogies while the latter tackles the issue from a different perspective and analyzes the cause behind the bad performances of traditional arc-consistency techniques in continuous domains.

2-B-consistency In [88], Lhomme proposes an interval propagation formalism based on bound propagation. A new consistency concept, weaker than arc-consistency and called 2-B-consistency, is introduced that assumes the convexity of variables' domains. In the case

where the effective domains are disjunctive, 2-B-consistency will consequently admit local inconsistencies within the labels. In Lhomme's approach, constraints are decomposed into primitive (basic) constraints allowing an easy definition of extrema functions. The relaxation step of the 2-B-consistency algorithm — analog to AC-3 — uses these extrema functions to refine variables' domains upon fixed-point iterate.

Box-consistency In [7], Benhamou *et al.* present a Prolog language (called `Newton`) extended to handle systems of non-linear equations and inequalities as well as constrained and unconstrained optimization. `Newton` implements a relaxed version of arc-consistency called box-consistency using the combination of interval arithmetic with a binary search technique. The goal of box-consistency is to determine an outer enclosing approximation of each variable's domain, as tight as possible. `Newton` implements the arc consistency algorithm AC-3 where each relaxation step enforces box-consistency on the unary projections of the revised constraint. Enforcing box-consistency amounts to identifying the left- and right-most quasi-zeros (a zero that cannot be distinguished from a zero due to machine precision) on an interval function. The search for a quasi-zero uses the main step of Newton interval method.

Since the Newton interval method is not sufficient to guarantee box-consistency, `Newton` applies an internal splitting operation focusing on parts of the intervals to locate the leftmost and rightmost quasi-zeros: each interval is split recursively into two other intervals of same sizes. If no quasi-zero is found in the left part, this part is simply pruned and the algorithm is restarted on the right part.

Both 2-B and box-consistency assume the convexity of the variables' domains and focus on optimizing the tightening of the feasibility space outer bounds. Their principal difference lies in the fact that the former requires interval tools for computing the unary projections of constraints while the latter works directly on the original constraints and approximates the projections using interval Newton iterates. Note finally that `Newton` generally produces sharper results than 2-B-consistency and is able to handle larger problems since no decomposition and no additional variables are required. It however imposes more restrictive applicability conditions (diagonally dominant Jacobians).

Arc-consistency using total constraints In [44], Faltings shows that some undesirable features of propagation algorithms with interval labels must be attributed to the inadequacy of the propagation rule and to a lack of precision in the solution space description. Faltings demonstrates that by using a network of total constraints along with a relaxation rule — based on the identification and classification of local extrema — sound and locally complete propagation becomes possible. Faltings arc-consistency algorithm is defined for binary constraints. Generalization to higher arity constraints has been implemented but requires more sophisticated mathematical preprocessing of constraints.

Discussion Since arc-consistency techniques enforce a weak degree of consistency, situations where no significant refinement of the variables' labels is obtained are unavoidable. Fixed-point algorithms are in effect sensitive to the *variable dependency* problem which engenders a loss of precision in the refinement process. The variable dependency problem refers to the fact that when the same variable occurs in two distinct constraints, the refinement obtained through fixed-point iterates is generally less precise than if the two constraints where synthesized prior to relaxation into a single one — for example analytically. Both Benhamou *et al.* and Faltings' algorithms work on the original constraints of the problems rather than on primitive constraints. This explains why the approximations they provide are sharper than those delivered by the standard implementations presented above (CLP(BNR) [8], etc.).

In the presence of *cyclic* constraints network, fixed-point iterations cannot guarantee convergence. Lhomme addresses the problem by introducing the notion of 2-B(w)-consistency — a partial form of 2-B-consistency — characterizing the degree of consistency obtained when the refinement process terminates abnormally (looping problems). w characterizes the imprecision of the computed bounds and 2-B(w)-consistency is equivalent to 2-B-consistency when $w = 0$. Using 2-B(w)-consistency, the complexity of the consistency technique can also be tuned by fixing the precision desired for the resulting interval bounds. In [44], Faltings stresses the need for algorithms computing labels of higher degrees of consistency to avoid the problem of infinite iterations.

2.5.5 Algorithms for higher degrees of consistency in continuous domains

Path-consistency algorithms such as PC-1 and PC-2 have been applied successfully to some restricted classes of continuous constraints (temporal constraints [38] and spatial constraints [132]). In the general case however, and in the absence of reliable tools for combining numerical constraints, it is difficult to reach even arc-consistency. Thus, only a few authors have investigated the issue of algorithms for higher degrees of consistency in continuous domains.

Tolerance propagation One of the first attempts, by Hyvönen [75] introduced the notion of *tolerance propagation* (TP) which generalizes the idea of numerical exact propagation into interval propagation. In the TP approach, numerical problems are represented as spreadsheet tables where each variable is initially set to an arbitrary interval. A forward chaining rules schema propagating solution functions is then used that refines the intervals further by determining the least common general solution for the problem. Elementary solution functions are evaluated using interval arithmetic. A local tolerance propagation algorithm, closely related to the Waltz filtering algorithm, is proposed that can be generalized into global TP for determining globally consistent solutions. The idea behind global tolerance propagation is to use global solution functions instead of the local ones during propagation. Global solutions functions can be theoretically evaluated using algebraic manipulations or interval arithmetic. The author proposes the combination of numerical and algebraic techniques for computing the actual value of an interval function (extension), and applies interval arithmetic algorithms for determining the most optimal extension. Since the evaluation of complicated interval functions is often computationally expensive, the author also proposes a partial globalization schema producing more optimal solutions than local tolerance propagation with less computations than global tolerance propagation. The idea here is to determine global solution functions only with respect to some critical variables and some sub-networks of the original constraint network.

k-B-consistency Lhomme [88] generalizes the notion of 2-B-consistency to K-B-consistency. K-B-consistency algorithms are proposed that can reach higher degrees of consistency than 2-B-consistency but refine uniquely the outer bound of the feasibility

domains.

2.6 Summary

As mentioned in the introduction of this thesis, many practical problems need to reason about ranges of feasible values and thus, require tools for constructing the most exhaustive description as possible of the feasible space. In this chapter we have shown that, among the numerous methods devoted to numerical constraint checking, only a few fit these requirements. In effect, methods for computing *pointwise solutions* even if efficient (numerical iterations, stochastic algorithms, linear and non-linear programming techniques) are directly precluded, while algebraic procedures have a too limited scope of applicability if used in isolation. Interval arithmetic and constraint satisfaction techniques — often used in conjunction — are more appropriate. Yet, important issues remain unexplored in interval-based consistency techniques and further research is necessary to render them suitable to practical use.

Chapter 3

Representing Continuous Constraints

"Most applications of quantitative inference support a locality assumption that there is natural grouping of quantities" [34]

A key to reliable consistency algorithms is the choice of a proper representation for constraints. The vast majority of consistency techniques developed for solving constraint satisfaction problems are restricted to discrete domains where representing constraints is simple. However, a large variety of practical applications, such as design, spatial and temporal reasoning, constraint based-graphics, configuration problems and numerous others, do not lend themselves to an arbitrary discretization. Research into the application of constraint-based techniques for these kinds of problems has led to the realization that the choice of a proper representation for continuous constraints is not a trivial issue [44]. In this chapter, we present a partially discretized representation of continuous constraints based on a hierarchical decomposition of the solution space. This representation avoids the drawbacks of standards representations while preserving the ability to deal with *ranges* (intervals) of feasible values.

3.1 Constraint and label representation

In the case of discrete CSPs, the enumerability of the solution space considerably simplifies the choice of constraints and labels representations. Labels are sets of discrete values representing possible instances for variables and constraints express the compatibilities (legal value combinations) between the values, either explicitly (matrices) or implicitly (arith-

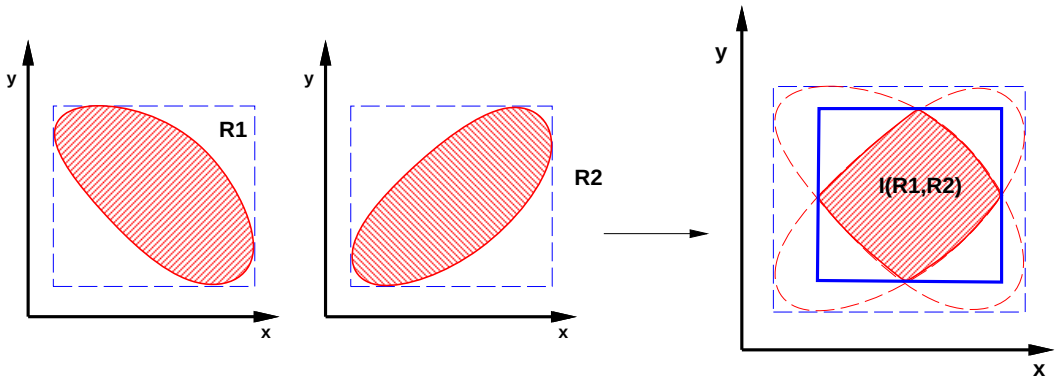


Figure 3.1: *The enclosing rectangle of an intersection of regions R_1 and R_2 is in general different from the intersection of the enclosing rectangles of R_1 and R_2*

metic formulas, boolean expressions etc..). The non-enumerability of continuous domains precludes a priori explicit representations. Fortunately, the following two considerations strongly mitigate this assumption:

- variables' domains have a naturally strict order
- continuous constraints have geometric interpretations providing a natural means of grouping values into aggregated entities

One of our main goals in this work was to investigate how these specificities of continuous solution spaces can be exploited to define expressive, reliable and yet practical representation of continuous constraints.

3.1.1 Interval-based approximations

Constraints are most naturally represented by algebraic or transcendental equalities and inequalities, described intensionally. Even if the accuracy of the solution space description is preserved, this representation does not allow an easy combination of constraints, thus precluding proper implementations of algorithms for higher degrees than arc-consistency. Mathematical methods compute the combined effect of simultaneous constraints using algebraic manipulation, for example Gaussian elimination, or iterative numerical refinements such as Newton, steepest descent, and gradient methods. When arbitrary constraints are involved, these techniques do not guarantee satisfactory solutions and often require careful modeling and much human guidance.

To overcome the difficulty of handling simultaneous constraints, constraint satisfaction methods with interval labels generally work with the *unary projections* of constraints rather than constraints themselves:

i. each variable is provided by a single enclosing interval, its *label*

ii. the feasibility space defined by a constraint is roughly approximated by an enclosing box whose dimensions are given by the intervals for the involved variables

Although this representation has the advantage of drastically reducing the number of possible solution regions for a variable (thus reducing the combinatorial effects), it suffers from the following deficiencies:

- the accuracy of the geometric information provided by constraints is greatly weakened
- pointwise relations between pairs (or sets) of variables are lost
- unfeasible values are admitted within the intervals for variables
- there is no way to distinguish between single and multiple solutions
- the solution space resulting from the combination of two constraints cannot be determined properly (see Figure 3.1)

Representations admitting several intervals per variable are sometimes used, for example in order to deal with disjunctive constraints, but the possible exponential splitting of intervals precludes this solution for general cases.

3.1.2 Incomplete propagation and cycles

The difficulty of combining simultaneous constraints has several undesirable consequences on interval-based consistency techniques. In contrast to discrete CSPs, network representations of continuous CSPs do not assume a single arc or hyper-arc between the same variables (see Figure 3.2)).

Indeed, when several simultaneous constraints link a set of variables — and since their combined effect cannot be computed easily in the most general case — an arc is introduced for each constraint in the network representation. Each arc is then handled

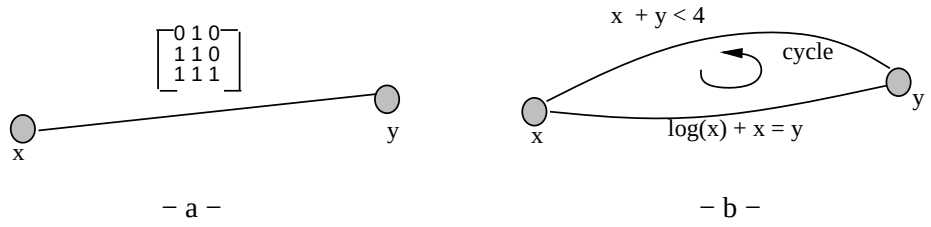


Figure 3.2: *In continuous domains, simultaneous constraints between the same variables lead to the creation of cycles in the network representation (**b**). This is not the case for discrete problems which generally assume a single arc or hyper-arc between a set of variables (**a**)*

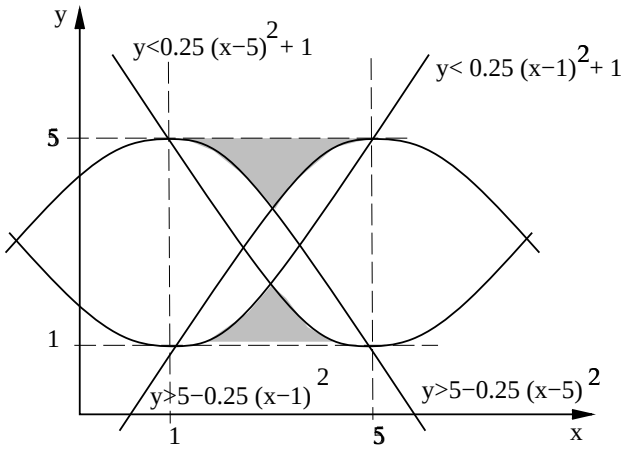


Figure 3.3: *When each constraint of a cycle is propagated individually, situations occur where simple inconsistencies cannot be detected*

individually during the propagation process, which leads to incomplete propagation [44]. This phenomenon, which might lead to a total absence of labels narrowing even for easy cases, severely affects the efficiency of propagation algorithms and rules them out as non-existence proof constructors for Constraint Logic Programming languages. Figure 3.3 (borrowed from [44]) gives an example where classical local propagation techniques fail to detect an inconsistency in a set of four simultaneous constraints between two variables.

In addition to the problem of premature quiescence, the existence of multiple arcs between the same variables create *cycles* in the network, which represents a major problem for propagation techniques, notably those based on the fixed point iteration schema. Cycles may lead to self-dependent refinements of the labels, thus precluding quiescence when locally convergent mechanisms are used by means of narrowing operators. In practice,

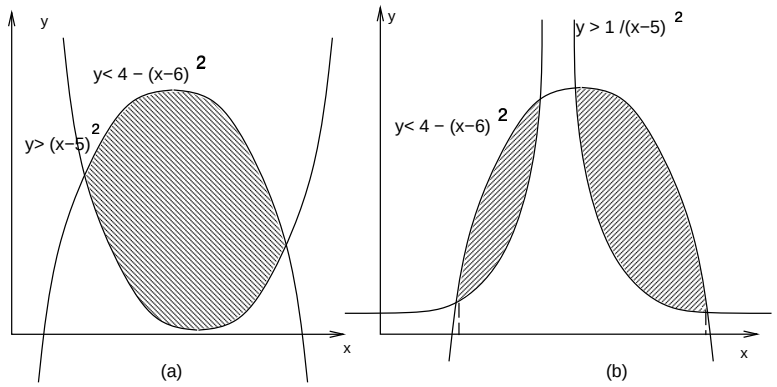


Figure 3.4: *The shaded regions ($\subset \Re^2$) represents the total constraints between x and y for two CSPs, (a) and (b), defined respectively as follows: (a)= $\{y > (x-5)^2, y < 4-(x-6)^2\}$ and (b)= $\{y > 1/(x-5)^2, y < 4-(x-6)^2\}$.*

self-dependencies do not produce infinite number of refinements since the limited accuracy of computer arithmetic will eventually force an end to the process. However these dependencies by engendering numerous refinement steps, may have a strong influence on the speed of the propagation process — which, in practical systems, is often stopped artificially after an arbitrary number of refinements.

These problems led us to investigate alternative techniques for both representing and handling constraints.

3.2 The 2^k-tree representation of constraints

We have seen that even when constraints are formally represented by mathematical expressions in traditional consistency techniques, they are handled in fact as rough enclosing box approximations of the effective solution space. In order to define a more precise and yet efficient representation of continuous constraints, we observe that most practical applications satisfy the following two assumptions:

- each variable takes its values in a bounded domain (bounded interval)
- there often exists a maximum precision with which results can be used

Provided that these two assumptions hold, a constraint $C_{x_1,\ldots,x_k}$ can be *approximated* by carrying out a hierarchical binary decomposition of its solution space into 2^k-trees (quadtrees for binary relations, octrees for ternary ones etc...).

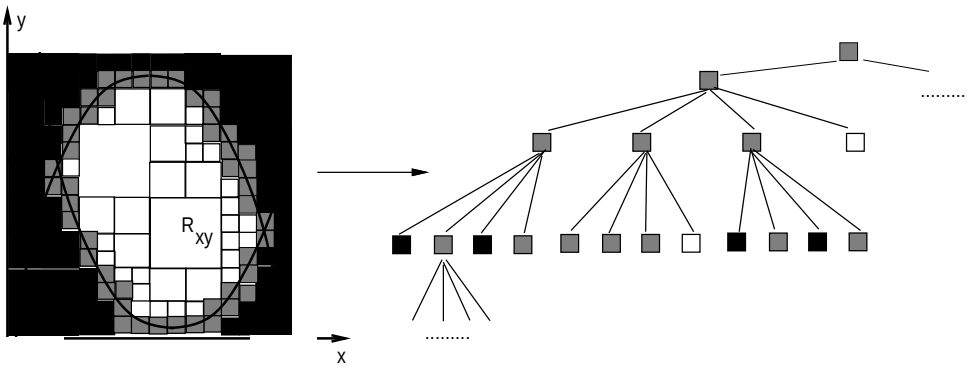


Figure 3.5: *A binary relation R_{xy} can be approximated by carrying out a hierarchical binary decomposition of its solution space into a quadtree*

Before describing this representation in more detail, we start by giving some needed definitions:

Definition 3.1 *:*

i. A ***continuous constraint*** $C(x_1 \ldots x_n)$ *is an arbitrary equality or inequality involving variables* $x_1, \ldots x_n$ *(* $x_1, \ldots x_n \in D_1, \ldots D_n$ *and* $D_i \in \Re$*).*

ii. A ***relation*** $R(x_1 \ldots x_n)$ *is the solution space defined by a set of constraints* $C_1, C_2 \ldots C_k$ *between the variables* $x_1, x_2, \ldots x_n$*.*

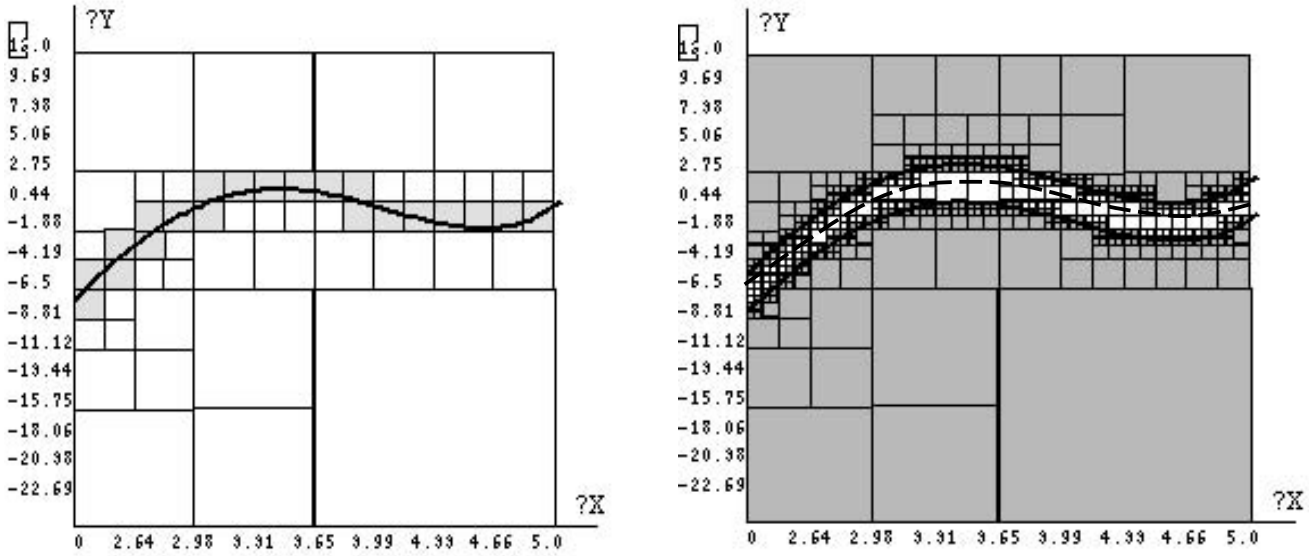
iii. A ***total constraint*** *between a set of variables* $x_1, \ldots x_n$ *is the relation including the whole set of continuous constraints involving* $x_1, \ldots x_n$ *in a given CSP.*

Figure 3.4 shows a total constraint between variables x and y for two binary CSPs.

3.2.1 Representing inequalities

A relation determined by *inequalities* can be approximated by a 2^k-tree (see Figure 3.5) where each node represents a k-dimensional cubic sub-region of the original domain (i.e., the domain over which the decomposition is carried out). A node has one of three possible states:

- *white:* if the region it defines is completely legal
- *gray:* if the region is partially legal and partially illegal



–A– –B–

Figure 3.6: *An equality constraint can be approximated either by the boundary nodes (gray nodes) of a 2^k-tree decomposition (**-A-**) or using the representation of a pair of enclosing inequalities (**-B-**).*

- *black:* if the region is completely illegal

When a black or white node is identified, the recursive division stops. Each gray k-dimensional cube is decomposed into 2^k smaller ones whose sides are one half the length of the original cube. Unless the boundaries of a region are parallel to the coordinates axes, infinite levels of representation are required to accurately represent a region. However, since the maximum precision is fixed, any gray node with a smaller size than the maximum granularity can be declared black and the decomposition stops.

3.2.2 Representing equalities

In the case of *equality* constraints, a strict application of the binary decomposition into a 2^k-tree as described would amount to pursuing the decomposition to infinity since an infinite degree of precision is required to represent single point solutions. However, many practical applications require a limited degree of precision, thus it is admissible to treat equalities with a certain error range.

There exist two ways of representing equalities using 2^k-trees:

- Since we observe that the boundary nodes of a 2^k-tree decomposition provide a good approximation of the solution space defined by an equality constraint (see Figure 3.6, -A-), one alternative is to use gray nodes of maximal resolution to represent equalities. In this case, gray nodes can be declared white and will represent solutions with error ranges.

- A strict equality constraint can be translated into a weaker form, $f(x_1, \ldots x_k) = C \pm \varepsilon/2$, where ε is the fixed maximum precision, as defined for inequalities. This amounts to replacing each equality by two inequalities (see Figure 3.6, -B-).

In both cases, the representation has the advantage that:

- the geometric information captured is far more expressive than standard enclosing box approximations

- a unified framework for handling both equalities and inequalities is provided

We discuss later in this chapter, the advantages of one representation over the other in terms of soundness issues.

As for other enclosing approximate representations, the 2^k-tree decomposition of constraints does not allow for an exact treatment of equalities. Some authors [134] address the problem by removing as many equalities as possible from the problem using algebraic manipulations for example.

3.2.3 Completeness and Soundness

The 2^k-tree representation of constraints provides two possible approximations of a solution space $\mathcal{S}$:

- the inner content approximation, $\mathcal{I}(S)$, is given by the white nodes (interior nodes) and is entirely enclosed within the solution space. In the case of inequality constraints, this approximation defines a *non-conservative* but *sound* set of feasible values,

- the outer content approximation, $\mathcal{O}(S)$, is given by the union of the white and gray nodes (interior nodes $\cup$ boundary nodes). This approximation is *complete* but *not sound* in the strict mathematical sense, since spurious values (within the regions delimited by gray nodes) are considered as feasible.

In the case where the 2^k-tree expansion has been carried out until the maximum precision required by the application is reached, we will consider that $\mathcal{I}(S) \equiv \mathcal{S}$.

The choice of one approximation rather than another plays a central role when numerical values condition decision processes. Sound approximations guarantee that no useless exploration of infeasible regions will be done and that no effort will be wasted in computing irrelevant information. On the other hand complete approximations guarantee that no solution is lost.

For many complex under-constrained problems with large and non enumerable solution spaces, the concern about the omission of marginal possible solutions might be outweighed by the increased inferential efficiency gained by sparing unintended inferences via incorrect assumptions.

We see the question of choosing the right type of approximation more as a dynamic issue depending on efficiency requirements and on the nature of the decisions involved: rough approximations can be constructed with little computational effort and are useful for coarse reasoning. In this case, choosing a conservative representation may compensate the weak expressiveness of the approximation. Further consistency checking can be carried out dynamically when decisions involve undetermined subparts of the solution space (gray nodes). Such an approach has been tested successfully in complex robotics problems consisting of computing qualitative decisions from sensor data [58] — the difference with our approach is that the dynamic refinements are operated using interval bisection and can be slow due to the inability of intervals to represent pointwise dependencies among components of an interval vector. In the case of 2^k-trees, the additional consistency checking are materialized by further subdivisions of gray nodes — a simple, deterministic and reliable process.

In order to bound the refinement process, we consider that after a given number of iterative refinements, the maximum precision required by the application is reached and we only keep the inner-content approximation as base for further inference.

Note finally that when the effective inner content reduces to a point set (equality constraints), and when equalities are approximated by two enclosing inequalities, even the $\mathcal{I}(S)$ representation (white nodes only) encloses the effective solution space. In this case, $\mathcal{I}(S)$ represents solutions with error ranges.

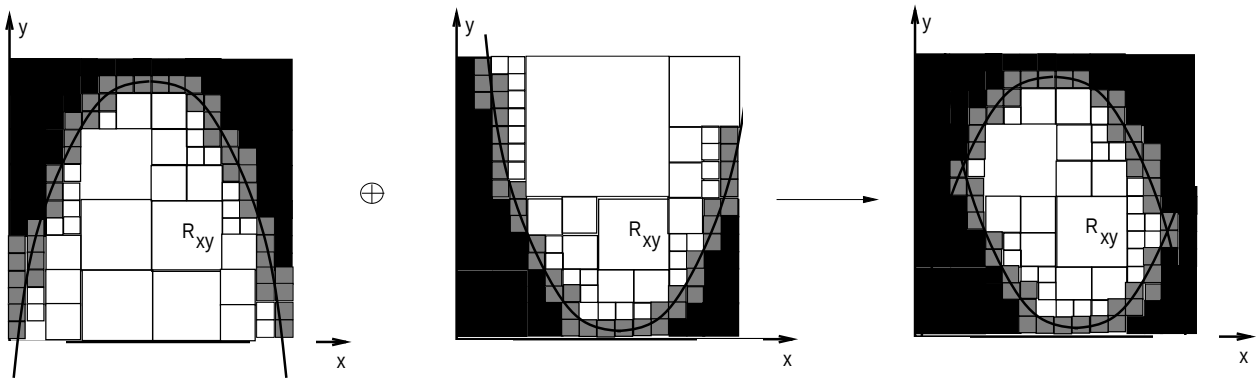


Figure 3.7: *A total constraint between a pair of variables is constructed by superimposing the quadtree representations of all individual constraints among these variables.*

3.3 Computing total constraints

The domain for each variable is fixed once and for all before constructing 2^k-tree approximations. Consequently, the discretization boundaries for a variable are the same for each constraint in which it participates. This simple convention provide a powerful means to combine constraints logically.

As illustrated in Figure 3.7, a total constraint between a pair (or set) of variables can be constructed by superimposing all the individual constraints among these variables. Superimposition (or intersection) is simply defined as the occlusion of feasible regions by infeasible ones. The intersection of two $2k$-trees (denoted by $\oplus$) is therefore implemented by combining logically the colors of each pair of corresponding nodes (i.e., nodes representing the same region in space).

Given an ordering $white < gray < black$, the rule for determining the feasibility of a node obtained by intersection is expressed as:

$$color(node_1 \oplus node_2) = Max(color(node_1), color(node_2)) \tag{3.1}$$

The algorithm of Figure 3.8 constructs the 2^k-tree approximation $A_{x_1,\ldots x_n}$ of a total constraint $C_{x_1,\ldots x_n}$, given intentionally by a set of equations and inequations $(C_1 \ldots C_l)$. Step 2 of this algorithm will be described in section 3.4.

Computing total constraints amounts, in the case of equalities, to approximating the solutions of systems of equations, sharing the same variables.

The simple example of Figure 3.9 shows the total constraints computed for two closely

Compute-total-constraints$(C_1, \ldots C_l)$

1 **foreach** $C_i \in (C_1 \ldots C_l)$ **do**
2 *Build a* 2^k*-tree representation* $A^i_{x_1,\ldots x_k}$ *for* C_i
3 $A_{x_1,\ldots x_k} \leftarrow A_{x_1,\ldots x_k} \oplus A^i_{x_1,\ldots x_k}$

Figure 3.8: *Algorithm for computing total constraints.*

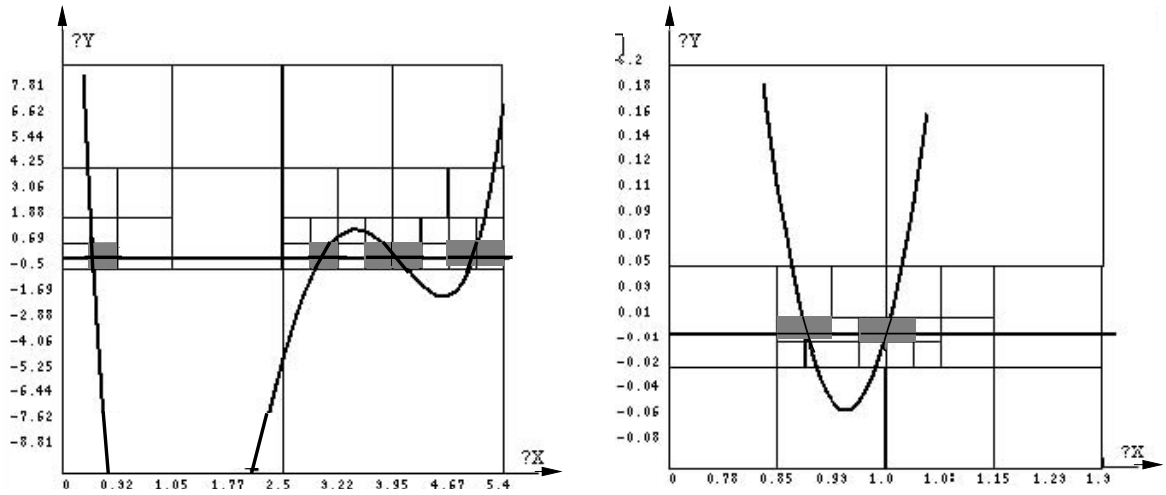


Figure 3.9: *An example where total constraints provide the approximate zeros of two closely related polynomial functions.*

related systems of constraints [77]:

$$\begin{array}{l} y = x^4 - 12x^3 + 47x^2 - 60x \wedge y = 0 \\ y = x^4 - 12x^3 + 47x^2 - 60x + 24 \wedge y = 0 \end{array} \tag{3.2}$$

In this example, total constraints provide the approximate zeros of two polynomials. They are constructed by intersecting logically the individual representations of $y = x^4 - 12x^3 + 47x^2 - 60x$, $y = x^4 - 12x^3 + 47x^2 - 60x$ and $y = 0$.

Correctness of the calculated representation As seen in section 3.2.3, the 2^k-tree representation theoretically provides two possible approximations of the effective solution space delimited by a set of constraints — namely the $\mathcal{I}(S)$ and $\mathcal{O}(S)$ approximations.

While the $\mathcal{I}(S)$ and $\mathcal{O}(S)$ approximations are computed *exactly* for individual constraints (using equation solvers), we have seen that the solution space for *sets* of simultaneous constraints is obtained through *logical* operations on 2^k-trees — the goal being to avoid complex and unstable numerical tools. One important question that presents itself is whether the logical operations on 2^k-trees preserve the correctness of the representation for sets of simultaneous constraints. In other words, whether the following assertion is verified:

$$2^k\text{-tree}\ (R_1) \oplus 2^k\text{-tree}\ (R_2) = 2^k\text{-tree}\ (R_1 \oplus R_2) \tag{3.3}$$

where 2^k-tree (R_i) denotes the 2^k-tree representation of a continuous relation R_i.

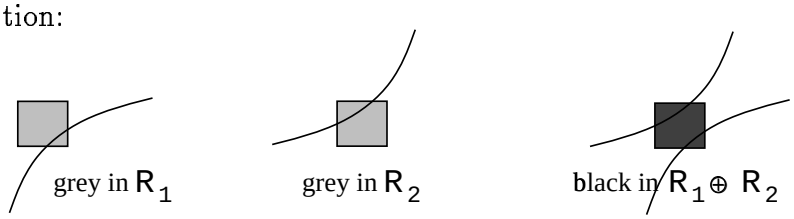
In the following, we will denote as $\mathcal{I}^*(S)$ and $\mathcal{O}^*(S)$) the inner and outer content approximations computed by logical combinations of simultaneous constraints.

Proposition 3.1 *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ the solution spaces of two relations R_1 and R_2 involving the same variables. We have: $\mathcal{I}(\mathcal{S}_1) \oplus \mathcal{I}(\mathcal{S}_2) = \mathcal{I}(\mathcal{S}_1 \oplus \mathcal{S}_2)$ (i.e. $\mathcal{I}(S) = \mathcal{I}^*(S)$).*

Proof. By definition, a white node corresponds to a *totally feasible* solution region. A node is therefore white in the representation of $R_1 \oplus R_2$ if and only if it is also white in the representations of R_1 and R_2. Hence, since the $\mathcal{I}(S)$ approximation of a relation contains only white nodes, the result is proved □.

Proposition 3.2 *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ the solution spaces of two relations R_1 and R_2 involving the same variables. We have: $\mathcal{O}(\mathcal{S}_1 \oplus \mathcal{S}_2) \subseteq \mathcal{O}(\mathcal{S}_1) \oplus \mathcal{O}(\mathcal{S}_2)$ (i.e. $\mathcal{O}(S) \subseteq \mathcal{O}^*(S)$).*

Proof. Similarly to the case of the $\mathcal{I}(S)$ approximation, the set of white nodes in $\mathcal{O}(\mathcal{S}_1 \oplus \mathcal{S}_2)$ is stricly identical to the set of white nodes in $\mathcal{O}(\mathcal{S}_1) \oplus \mathcal{O}(\mathcal{S}_2)$. However, due to imprecision considerations, a node computed by intersection can be gray even if it contains no solution:



□.

These results guarantee that:

- the $\mathcal{I}^*(S)$ representation computed by logical combination of simultaneous constraints is complete and sound with respect to the effective inner content approximation $\mathcal{I}(S)$,
- the $\mathcal{O}^*(S)$ representation computed by logical combination of simultaneous constraints is complete but not sound with respect to the minimal enclosing approximation $\mathcal{O}(S)$ — spurious gray nodes can be created by logical intersection,

In the case of equalities, the representation based on two enclosing inequalities (see Figure 3.6 -B-) is therefore preferable. As opposed to the representation using boundary nodes (Figure 3.6 -A-), it maintains a clear distinction between the undetermined nodes and those guaranteed to contain at least one solution.

Cycles and early quiescence Almost all the existing interval-based methods approximate the combined effect of simultaneous constraints using a fixed point iteration scheme. Their convergence is consequently sensitive to the choice of proper initial conditions and the number of iterations they require is hard to predict, especially when the constraint network contains cycles.

The construction of total constraints illustrates how *explicit cycles* — those composed of several constraints between the same variables — are easily removed and highlights one major advantage of 2^k-tree representations.

Since total constraints are computed by logically intersecting the individual representations of constraints, the total absence of domains narrowing (or the infinite loops) provoked by the propagation of *individual* constraints involved in explicit cycles are avoided.

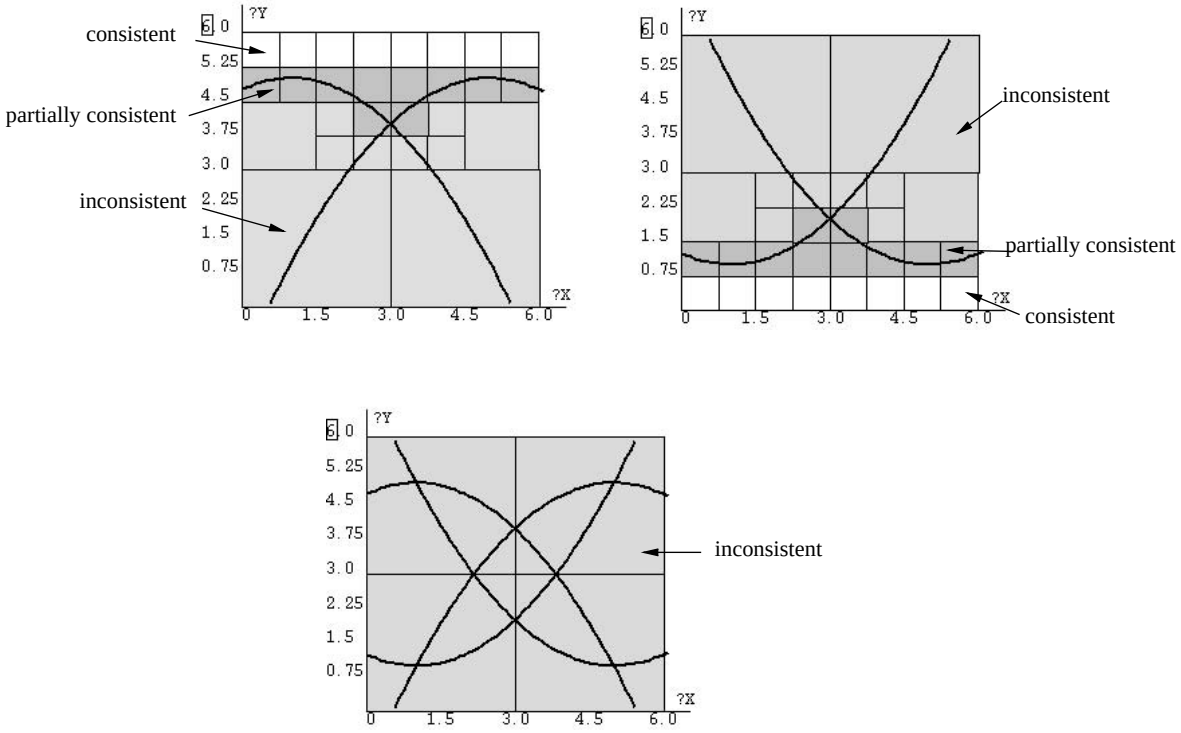


Figure 3.10: *When the solution regions for the system* $\{y < 0.25(x-5)^2 + 1 \wedge y < 0.25(x-1)^2 + 1\}$ *and for the system* $\{y > 5 - 0.25(x-5)^2 \wedge y > 5 - 0.25(x-1)^2\}$ *are combined logically using quadtrees intersection, the inconsistency in the constraints set* $\{0.25(x-5)^2 + 1 \wedge 0.25(x-1)^2 + 1\} \bigcup \{5 - 0.25(x-5)^2 \wedge 5 - 0.25(x-1)^2\}$ *is detected easily.*

For example, the inconsistency in the constraints set of Figure 3.3, is detected easily when the solution regions for the system $\{y < 0.25(x-5)^2 + 1 \wedge y < 0.25(x-1)^2 + 1\}$ and for the system $\{y > 5 - 0.25(x-5)^2 \wedge y > 5 - 0.25(x-1)^2\}$ are logically combined using quadtrees intersection (see Figure 3.10).

Implicit cycles — those constituted of constraints which can be transitively composed into a single constraint — will be discussed in more detail when tackling the issue of consistency techniques.

Constructing the 2^k-tree representation of an individual constraint, requires a procedure for determining the color of each sub-region (k-rectangle) created by the recursive decomposition. Two cases have to be considered:

i. if the constraint curve determines a *transverse segment* within the considered rectangle, testing for the rectangle color amounts to finding an intersection of its boundaries with the curve

ii. the curve is *closed* within the considered rectangle: a pre-processing phase must be carried out in order to split the curve into transverse segments

Figure 3.11 illustrates these notions in the case of binary constraints.
In order to simply identify the types of curves treated, constraints are divided into two broad categories: *functional* and *non-functional*. Functional constraints always determine open curves while non-functional might define closed ones. We perform the pre-processing phase systematically when non-functional constraints are involved.
Before discussing the general case of n-ary (with $n > 2$) constraints, first we present the technique implemented for binary constraints.

3.4 Constructing 2^k-trees

3.4.1 Binary constraints

Intersection test Determining the intersection of a rectangular sub-region of the domain with an open curve consists in solving systems of the form:

$$\{C(x,y), x = value, y_{low} < y < y_{up}\} \tag{3.4}$$

where $value$, y_{low} and y_{up} are constants defining a given side of the rectangle. Solving such a system amounts to finding the zeros of single equations with one variable. When $C(x,y)$

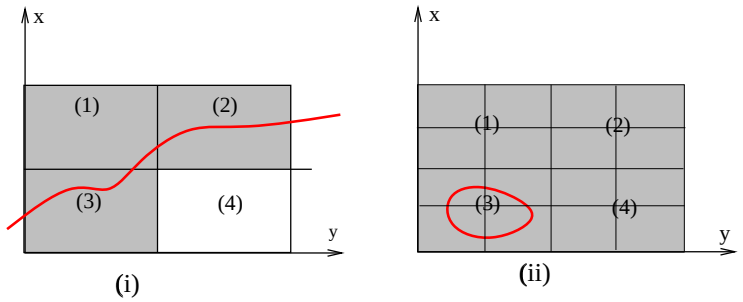


Figure 3.11: *In case (i), the constraint curve determines a transverse segment: if the boundaries of a rectangle intersect the curve , the rectangle is gray (rectangle(1)). If not it suffices to test if one vertex of the rectangle satisfies the constraint to know if it is white or black (rectangle (4)). In case (ii), a rectangle might be gray even if all of its vertices satisfy (or do not satisfy) the constraint (rectangle(3)). A preliminary search must be carried out to determine the divisions where the constraint curve is transverse.*

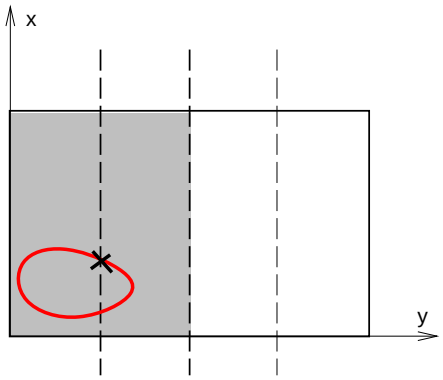


Figure 3.12: *The original approximation (largest rectangle) is initially undetermined. A binary search procedure is used for determining a division which intersects the constraint curve(line with a cross). The rectangle enclosing the division (the shaded one) is then known to be gray and can be used as initial approximation domain*

is functional in y the task reduces to a simple evaluation. Otherwise, there are a wide variety of mathematical techniques which propose efficient solutions to the problem. When arbitrary constraints are involved, we resort to iterative numerical analysis techniques. In the case of binary constraints, a maximum of four tests — one for each side — are required for determining the color of a rectangle. Note finally that the task of solving the system $\{C(x,y), x = value\}$ is performed only once and for all, for each $x = value$. For all the rectangles having a side along the line $x = value$, the intersection test then reduces to check whether a solution to $\{C(x,y), x = value\}$ falls within the considered boundaries, y_{low} and y_{up}.

Preprocessing of closed curves Closed curves require a more sophisticated handling since the color of the initial approximation cannot be determined by a simple intersection test (see Figure 3.11). The role of the preprocessing phase is to check whether a solution exists within the initial approximation. In this work we only consider the case where a constraint determines *one* single closed curve. We are not aware of practical cases where a single, non-disjunctive, continuous constraint generates multiple closed regions. Disjunctive constraints are known a priori and each disjunctive alternative can be treated as an individual subproblem.

The ambiguity concerning the color of the initial approximation is removed by identifying a point of the domain belonging to the curve. This task can be achieved for example, by carrying out a binary search for determining a division which intersects the constraint curve (see Figure 3.12). The complexity of such a numerical search is reduced in practice by the fact that initial domains are the most often set to reasonable values and that a limited degree of precision is required.

3.4.2 n-ary constraints (n > 2)

In many realistic problems, the constraints are not binary, but n-ary. The presence of high arity constraints increases the complexity of solving CSPs. In the case of our 2^k-tree representation, this complexity is reflected by:

- the worst-case exponential growth of space requirements. An n-ary constraint is in the worst case represented by a 2^n-tree
- the mathematical complexity related to computing the color of a multi-dimensional node. Computing the color of a k-dimensional node amounts to finding the intersection of a k-dimensional rectangle with a $(k-1)$-D surface

For the case of discrete CSPs, a simple procedure exists for transforming n-ary problems into binary ones without losing any information [115]. This procedure is not available for continuous CSPs. Moreover, the approximation of n-ary continuous constraints by binary ones generally denatures the feasibility space, since projection introduces spurious solutions.

From the broad perspective of interval arithmetic, where the goal is to bound solutions with no particular concern about soundness, this projection approach remains an

alternative to consider for some category of problems.

Nevertheless, for uniformity and generality purposes, we have studied an alternative approach, more conservative with respect to the effective feasibility space.

Our approach is based on the fact that each n-ary constraint can be reduced to a set of ternary constraints without loss of information. A continuous n-ary relation, $C(x_1, \ldots x_n)$, can be transformed into a set of ternary algebraic expressions by:

i. replacing iteratively in C each sub-expression $< x_i operator x_j >$ by a new variable x_{n+1},

ii. adding a ternary equality constraint $x_{n+1} =< x_i operator x_j >$,

The process stops when C itself becomes ternary. This transformation is only based on symbolic manipulations and consequently no information is lost in the solution space description. For example, the 5-ary CSP with one constraint, $(x-y)^2 + \frac{(z+t)}{u} > 2$, can be translated into a ternary one with three constraints: $w_1^2 + (w_2/u) > 2$, $w_1 = x - y$, $w_2 = z + t$. The complexity of such a decomposition is in $O(m)$, where m is the number of operators. Hence, addressing n-ary continuous CSPs amounts to giving the ternary counterparts of the algorithms and representation used for solving binary continuous CSPs.

Intersection test Computing octrees for ternary relations can be carried out similarly to the case of quadtrees.

The difference lies in the fact that testing for a node color amounts in 3D to determining the intersection of the planar facets of the node with the surface determined by the relation. In the worst case, six intersection tests are necessary (one for each facet) instead of the four required for binary relations.

The intersection test amounts to solving systems of the form:

$$\{C_{xyz}, x = value, y_{low} < y < y_{up}, z_{low} < z < z_{up}\} \tag{3.5}$$

Various methods ranging from interval arithmetic methods([81, 82, 61, 60], etc.) and gridding methods [104] to random algorithms [134] offer efficient tools for solving the problem. Similar methods also apply to the preprocessing of closed curves in 2D. Our

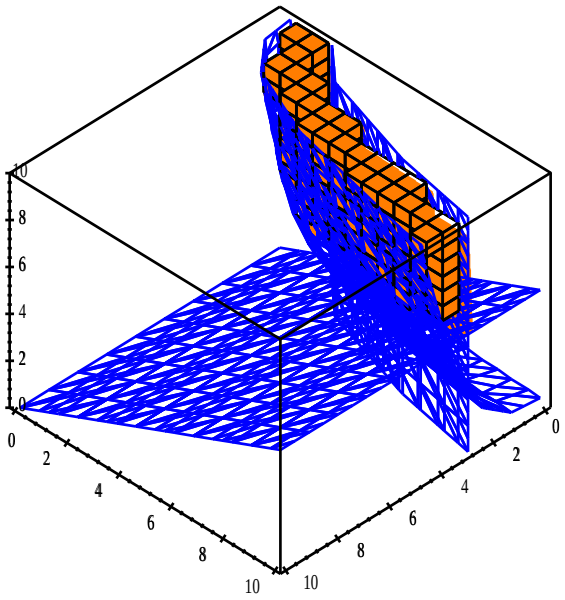


Figure 3.13: *A total constraint in 3D*

system implements the gridding method[1], chosen for its simplicity and generality rather than for optimization considerations. Figure 3.13 gives an example of total constraints in 3D (to aid clarity only the feasible nodes are displayed).

3.4.3 Complexity, sensitivity to initial conditions, iterative refinements and parallelization

2^k-trees intersection 2^k-trees can be encoded using linear structures which store only the feasible nodes rather than the whole set of possible nodes (including unfeasible and partially feasible ones). These structures will be described in more detail in the next chapter. Using a linear representation, intersecting two trees requires $O(N_1 + N_2)$ logical operations, where N_1 and N_2 are the number of feasible nodes in each tree [52].

2^k-trees construction In the worst case, the 2^k-tree resulting from the construction procedure is complete (each node has 2^k children). Let d be the maximum granularity (maximum number of recursive quarterings), the number of nodes in a 2^k-tree is given by

[1] the plane is decomposed into a grid and each section then examined

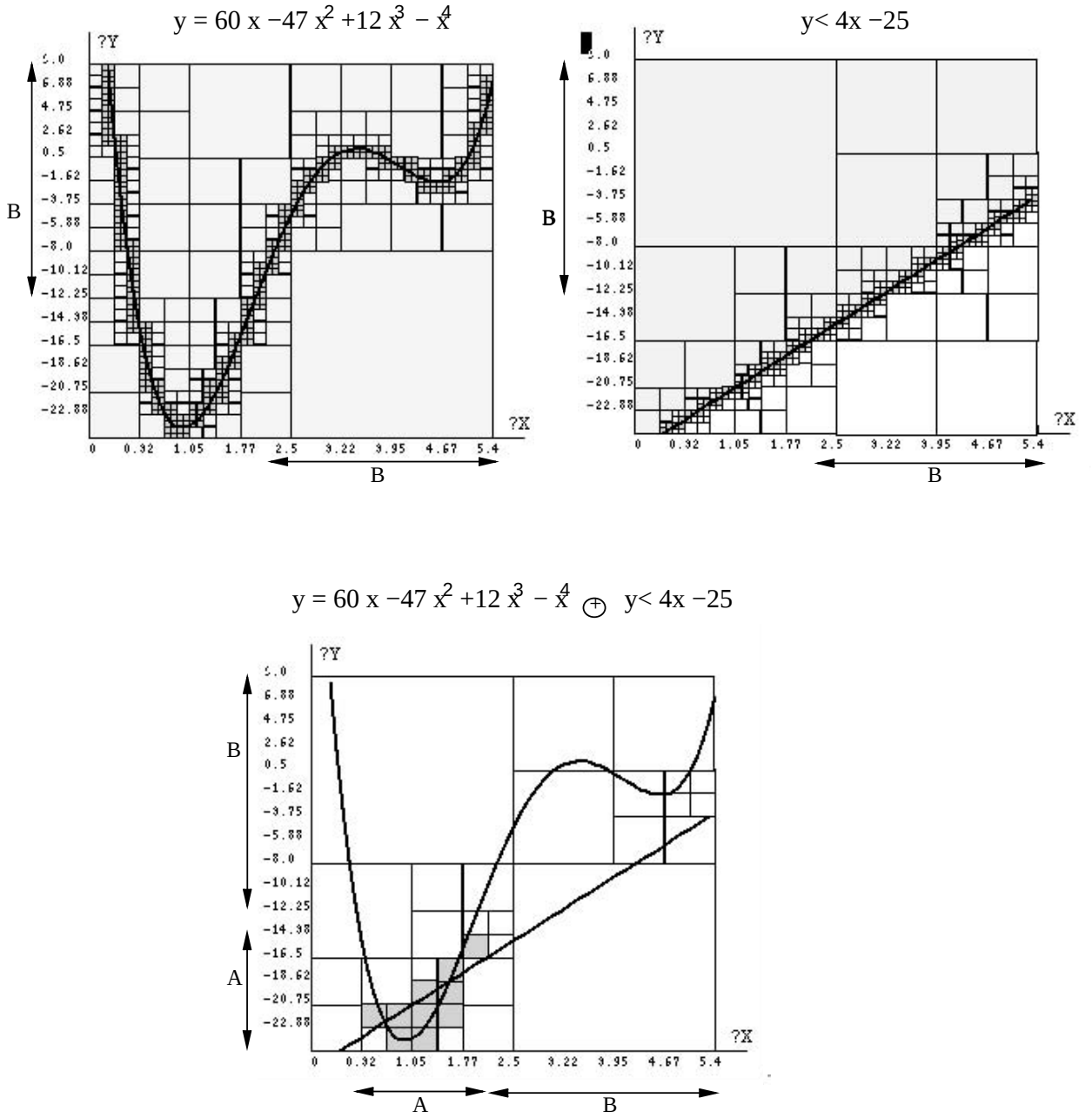


Figure 3.14: *Intersecting rough approximations (**bottom**) often aids in determining the subregions worth being finely decomposed. Combining intersection with 2^k-tree construction can spare the generation of numerous unnecessary nodes (**top**).*

Constraint	k	d	Real # of nodes	Expected # of nodes
$\sqrt{x^2+y^2} < 50$	2	6	688	5461
$\frac{12y}{\sqrt{x^2+y^2}} < 10$	2	6	340	5461
$y = x^2$	2	7	685	21845
$\cos(\phi_2 - \phi_1) > 0$	2	7	1492	21845
$x = r\cos\phi$	3	5	4740	37449
$y > 0.086x - 5.51e^{-5}x^2 + 8.36e^{-9}x^3$	2	5	115	1365
$z = -y - 9.63e^{-5}(0.041x)^{1.51}$	3	5	4117	37449
$x < (y+z)^{1/3}$	3	5	2857	37449
$x > 2/5(y+z)$	3	5	2787	37449
$x < \sin(\phi_2 - \phi 1)$	3	4	988	4681
$x < \frac{y}{\sqrt{z}}$	3	4	1107	4681
$x < y + z$	3	4	1072	4681

Table 3.1: *The number of nodes effectively computed while constructing a 2^k-tree is generally smaller by far than the worst case estimation.*

the following formula:

$$\sum_{i=0}^{d} 2^{k(d-i)} = \frac{2^{k(d+1)} - 1}{2^k - 1} \tag{3.6}$$

Hence, constructing a 2^k-trees roughly requires in the worst case $O(2^{k*d})$ where k is the arity of the constructed 2^k-tree and d the maximum granularity. For practical cases, d is given by the term s/ε where s is the maximum domain size (the largest interval size in D, for a CCSP (V, C, D)) and ε the tightest interval size accepted for variables of V. In the majority of cases, the measurements given above strongly overestimate the complexity. A more realistic measure should be done in terms of the number of gray nodes generated, since the recursive quartering stops as soon as a node color is set to white or black. More precisely, we can show that the number of gray nodes generated while decomposing a region is in $O(b^{k-1})$ [132] where b measures the boundary size of the represented k-dimensional region. When the boundary delimits sufficiently wide regions ($>> \varepsilon^k$), the number of gray nodes generated is less than $\frac{2^{k*(d+1)}-1}{2^k-1}$ by far. Table 3.1 illustrates this point on a sample of binary and ternary constraints derived from practical problems.

In general, computing total constraints however remains sensitive to the ratio between the maximal precision and the size of the approximation domain (for both construction and intersection tasks, since a fine decomposition increases the number of nodes).

Compute-total-constraints-iteratively $(\{C_1 \ldots C_l\}, \varepsilon)$:

1 *Choose a rough precision* ε',
2 **foreach** $C_i \in (C_1 \ldots C_l)$ **do**
3 *Build a* 2^k*-tree representation* $A^i_{x_1,\ldots x_k}$ *for* C_i *with precision* ε'
4 **od**
5 **repeat**
6 **begin**
7 $A_{x_1,\ldots,x_k} = A_{x_1,\ldots,x_k} \oplus A^i_{x_1,\ldots,x_k}, i : [1..l]$;
8 $D_j \leftarrow \prod_{x_j} A_{x_1,\ldots x_k}, j : [1..k]$;
9 **foreach** $A^i_{x_1,\ldots x_k}$ **do**
10 $A^i_{x_1,\ldots x_k} \leftarrow Refine(A^i_{x_1,\ldots x_k}) in\ D_j, j : [1..k]$;
11 **od**
12 **end**
13 **until** $precision(A_{x_1,\ldots x_n}) \leq \varepsilon$

Figure 3.15: *Algorithm for computing total constraints - iterative version.*

When the combined effect of constraints is considered, this sensitivity can be reduced substantially by resorting to iterative refinement of the hierarchical decomposition. In fact, since intersecting two 2^k-trees generally produces a restricted region with smaller projections over the axes, decomposing each individual constraint until the maximal precision is reached is unnecessary, and likely to generate many useless intersection tests. For example, decomposing to maximal precision the two constraints of Figure 3.14, prior to combining them, generates numerous useless nodes within region B. The idea behind iterative refinement of total constraints is to start from a rough approximation for each constraint, and refine each approximation only within the restricted region corresponding to their intersection (region A in Figure 3.14). The refinement of each approximation is obtained throughout further subdivision of their gray nodes, within the restricted region delimited by earlier combinations.

An iterative version of the algorithm of Figure 3.8 is given in Figure 3.15. The algorithm starts by computing, with a rough precision, the total constraint associated with a set of k variables (step 7). The domain for each variable is then refined by projecting over the individual axes involved the total constraint computed at step 7 (step 8). Finally (steps 9-12) , each individual constraint is refined within the domains tightened at step 8. The whole process is re-iterated until the desired precision is reached for the total constraint.

The complexity measures given above assume a sequential processing of the trees using a single processor. Since the operations described lend themselves naturally to a parallel processing, substantial improvements can be gained using multiple processors [52].

3.5 Topological properties and limit cases

While the 2^k-tree representation of constraints presents many advantages in terms of expressiveness, simplicity and reliability, a remaining issue that presents itself is whether this representation conserves the topological properties of the original solution space (connectivity, convexity etc...). Firstly, since the 2^k-tree decomposition generates stepwise

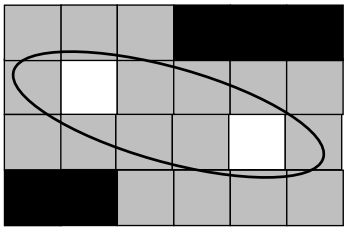


Figure 3.16: *When the resolution in insufficient, a connected solution region can be represented by disconnected nodes.*

approximations of the boundaries, convexity is obviously not preserved in the strict mathematical sense. Moreover, as shown in Figure 3.16, when the resolution chosen is insufficient, situations may occur where a connected solution space is represented by disconnected 2^k-tree regions.

We show in the following that when the original solution space is convex, the discretization process can only introduce restricted types of convexity deficiencies. We also show that the risk of disconnection only occurs when identifying the solution region requires reaching the limit of the resolution chosen for the 2^k-trees construction.

We start by giving some definitions:

Definition 3.2 *A neighbor of a node can be a node which is adjacent to the specified node or, a node which is positioned diagonally next to the specified node.*

Definition 3.3 *A 2^k-tree representation is connected if each pair of minimal size feasibility nodes can be joined by a path entirely contained in the feasibility region.*

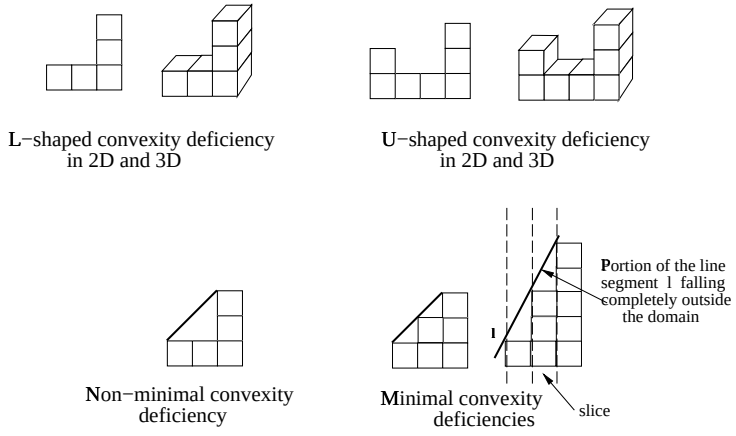


Figure 3.17: *Types of convexity deficiencies in quadtrees and octrees*

By construction, the elementary convexity deficiencies in a connected 2^k-tree approximation can be either L-shaped or U-shaped (see Figure 3.17). In a Cartesian coordinate system, we define a *slice* as an interval-line (respectively interval plane), orthogonal to a given axis x and of width ε_x , where ε is the minimal interval length for variable x in a quadtree (respectively octree) decomposition. We define:

Definition 3.4 *A convexity deficiency in a quadtree (respectively octree) is minimal if:*

i. it is L-shaped

ii. the two vertices (nodes' corners) delimiting the deficiency can be joined by a line segment l such that: each portion of l falling completely outside the feasibility region can be entirely contained in a single slice of minimal size (see Figure 3.17)

Now we are in position to state the following propositions:

Proposition 3.3 *Let $\mathcal{S}$ be a convex solution space. The $\mathcal{O}(S)$ representation of $\mathcal{S}$ is non-empty and connected.*

Proof. If $\mathcal{S}$ is non-empty within the original approximation domain, the $\mathcal{O}(S)$ representation contains at least one gray node: the one representing the approximation domain itself. Hence, $\mathcal{O}(S)$ is non empty. We assume now that $\mathcal{S}$ is convex. Suppose that $\mathcal{O}(S)$ is disconnected, this would mean that there exists within $\mathcal{O}(S)$ a pair of nodes n_1 and n_2 such that every path joining n_1 and n_2 traverses at least one black node. A fortiori, the

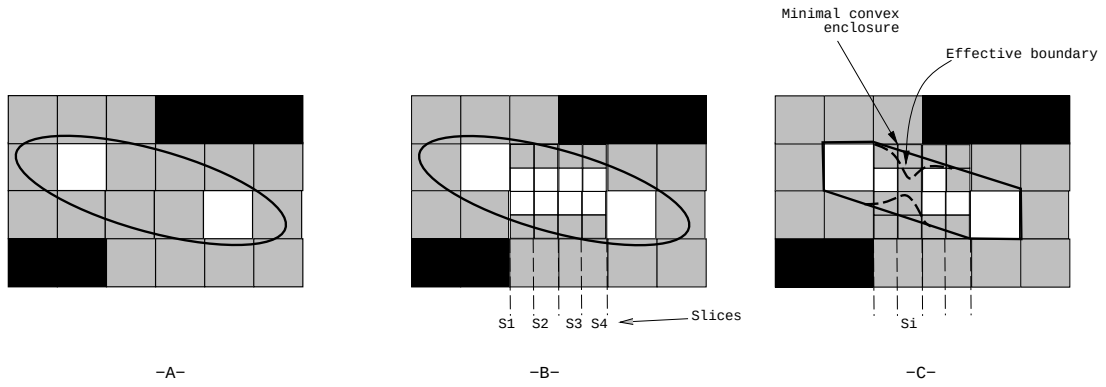


Figure 3.18: *If a convex solution space $\mathcal{S}$ yields a disconnected $\mathcal{I}(S)$ representation, such that the disconnected regions of $\mathcal{I}(S)$ are of minimal size, then one single additional level of decomposition suffices to reconnect the representation— elements of the proof.*

line segment $\overline{q_1 q_2}$, where q_1 is a feasible point of n_1 and q_2 is a feasible point of n_2, would necessarily traverse at least one totally unfeasible region. This contradicts the fact that $\mathcal{S}$ is convex □.

Proposition 3.4 *Let $\mathcal{S}$ be a convex solution space. If $\mathcal{S}$ yields a disconnected $\mathcal{I}(S)$ representation such that the disconnected regions of $\mathcal{I}(S)$ are of minimal size, then one single additional level of decomposition suffices to reconnect $\mathcal{I}(S)$*

Proof. Suppose that a convex region $\mathcal{S}$ yields a disconnected $\mathcal{I}(S)$ representation. According to propositions(3.3) each node of the disconnected area is gray.

We know that the disconnected regions are of minimal size (see Figure 3.18, -A-). Hence, for showing that one additional level of decomposition reconnects $\mathcal{I}(S)$, it suffices to show that each slice of size $\varepsilon/2$ (where ε is the minimal resolution) contains at least one white node (see Figure 3.18, -B-).

Suppose that there exists a slice s_i containing no white node of size $\varepsilon/2$. This would mean that the boundaries of the effective solution space traverse all the gray nodes of the slice, which implies that the effective boundaries fall below the minimal convex enclosure of the originally disconnected regions (see Figure 3.18, -C-). This contradicts the hypothesis that the solution space is convex □.

Corollary 3.1 *Let $\mathcal{S}$ be a convex solution space, its inner-content approximation $\mathcal{I}(S)$ can be disconnected only when the identification of the solution region requires reaching the limit of resolution*

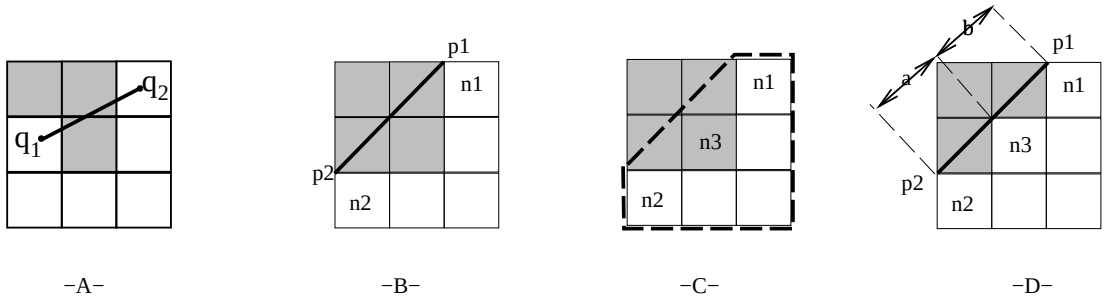


Figure 3.19: *The connected representation of a convex solution space only contains minimal deficiencies— elements of the proof.*

Proposition 3.5 *Let $\mathcal{S}$ be a convex solution space. If $\mathcal{I}(S)$ is connected, then it only admits minimal convexity deficiencies*

Proof. The existence of a U-shaped convexity deficiency in the $\mathcal{I}(S)$ representation of a convex region $\mathcal{S}$ contradicts the convexity hypothesis.

In case a U-shaped convexity exists, one could choose two feasible points, q_1 and q_2, within $\mathcal{I}(S)$ such that the segment $\overline{q_1 q_2}$ cuts the boundary of $\mathcal{S}$ (the gray nodes are by definition traversed by the boundary of $\mathcal{S}$) (see Figure 3.19, -A-). Suppose now that there exists a non minimal L-shaped deficiency. This means that there exist two nodes n_1 and n_2 such that the line segment $\overline{p_1 p_2}$ falls outside the feasible space and is contained in more than one slice (see Figure 3.19, -B-)

The minimal convex enclosure of the L-shaped deficiency is given by the dashed contour of Figure 3.19, -C- and contains the node n_3. Hence n_3 must be feasible.

The fact that n_3 is feasible means that each portion (a and b in Figure 3.19, -D-) of the segment $\overline{p_1 p_2}$ falling completely outside the feasible space, is entirely contained in one slice (see Figure 3.19, -D-). This would mean that the L-shaped convexity deficiency is minimal, which contradicts the hypothesis□.

We will show in chapter 5 that minimal convexity deficiencies are inconvenient only for limit cases where solutions exists under the maximal precision required by the application.

Discussion Discretizing the solution space using 2^k-trees inevitably introduces artificial convexity deficiencies. We have shown that these convexity deficiencies can alter the connectivity properties of the original feasible domain only in the case where the solution region falls within the limit of resolution. Furthermore, we demonstrate that a single

additional level of decomposition suffices to reconnect the feasible domain. In practice, the maximal resolution is fixed by the application and the solutions falling under this resolution can often be neglected. Hence, when a 2^k-tree presents disconnections and, in the case where an additional level of decomposition is not sufficient to reconnect the representation, the effective solution space can generally be considered as non-convex.

3.6 Discussion and summary

In computing the complete solution space for a CSP, one important problem is how to represent it concisely. In the worst case, the number of solutions to a CSP may be exponential in the number of variables and it is clearly not practical to give a list of all possibilities. When variable domains are continuous, there is the additional problem of how to discretize the continuum of values.

Consistency techniques, compactly represent the solution space using *labels* assigned to *individual* variables or *combinations* of variables.

One major limitation of standard interval-based techniques relates to the fact that the solution space is described using the labels of *individual* variables. This amounts to working with the unary projection of constraints rather than with constraints themselves and is not sufficiently precise for representing the effective labels associated with *combinations* of variables. For many practical problems, the solution space needs to be captured more accurately and a clear distinction made between variables domains and the solution regions delimited by constraints.

In this chapter we proposed how one could approximate regions of feasible solutions by 2^k-trees, a representation commonly used in vision and computer graphics and based on a hierarchical decomposition of space.

In contrast to standard interval-based representation, 2^k-trees allow the *explicit* handling of different solution regions. They provide simple means to combine constraints logically rather than numerically. This rules out the convergence problems related to the choice of proper initial conditions for constructing satisfactory approximations.

This chapter also discussed several issues relevant to the automation of 2^k-trees construction.

Chapter 4

Consistency Algorithms using 2^k-trees

Since it maintains pointwise relations between variables, the 2^k-tree representation of continuous constraints presented in the last chapter is more accurate than standard enclosing interval-based representations. Within the context of constraint satisfaction, the first step towards solving a CSP is to provide tools for pruning inconsistent regions of the solution space using the information provided by constraint combination. This chapter describes how such tools can be defined for continuous constraints using the 2^k-tree representation and presents how a logical rather than numerical handling of continuous constraints conveys a simple implementation for path- and higher degrees of consistency algorithms.

4.1 Introduction

k-consistency algorithms [46, 90, 99] compute consistent labels for each subset of k variables involved in a problem. Intuitively, the label for any subset of $k-1$ variables admits only those value combinations such that for any additional k^{th} variable, there is at least one value consistent with these $k-1$ assignments. More precisely, computing the label for the set of variables $x_1, x_2, ..., x_k$ requires the definition of the following two operations on constraints [46]:

- *computing induced constraints:* which consists of determining implicit higher dimensional constraints involving variables $x_1, x_2, ..., x_k$,
- *relaxing constraints:* which consists of revising any explicit constraint between $x_1, x_2, ..., x_k$, given the influence of all induced constraints involving these variables

Constraint relaxation is implemented as the *intersection* ($\oplus$) of an existing constraint with all *induced* constraints involving its variables. An induced constraint is given by the *composition* ($\otimes$) of a pair of $(k-1)$-ary labels, subsets of the k variables for which the label is being computed. Finally, the set of k-ary labels must be *projected* ($\prod$) back into the $(k-1)$,$(k-2)$,...,1-ary labels.

As mentioned in chapter 3, when constraints are represented using intensional mathematical descriptions, the implementation of the intersection, composition and projection operators requires reliable tools for solving arbitrary sets of simultaneous equations and inequations. In the absence of such tools, the existing interval-based techniques restrict themselves to working with the unary projection of constraints, which is insufficiently precise for implementing path-consistency — and more generally, algorithms for higher degree of consistency than arc consistency.

This explains why algorithms enforcing path-consistency, and *a fortiori* higher degrees of consistency, are rarely used in continuous domains. The work of Dechter *et al.* [38] on simple temporal constraints constitutes one of the few exceptions. In simple temporal problems, constraints take the form of bounded differences $b_1 < x_i - x_j < b_2$ where $[b_1 b_2]$ has to be a *single* interval. This restriction makes it easy to implement the composition and intersection operators required for path-consistency, which then reduce to modification of scalars.

By handling explicitly the values of feasible regions, the 2^k-tree representation provides an easy and reliable means of combining simultaneous constraints. The rest of this chapter is devoted to showing that, equipped with this explicit representation of solution spaces, it becomes possible to design reliable consistency algorithms transferred from the area of discrete CSPs.

We start by giving some needed definitions.

Definition 4.1 *:*

i. A continuous CSP *(CCSP), (V, C, D), is defined as a set V of* ***variables*** *$x_1, x_2, \ldots x_n$, taking their values respectively in a set D of continuous* ***domains*** *$D_1, D_2, \ldots, D_n$ and constrained by a set of* ***relations*** *$C = \{R_1, \ldots, R_m\}$.*

ii. A ***domain*** *is an interval of $\Re$.*

iii. A ***relation*** *is defined intensionally by a set of arbitrary equalities and inequalities.*

Given a CCSP and according to the last above definition, we require that for each subset of variables $(x_1, \ldots x_k)$, a unique relation $R_{x_1 \ldots x_k}$ exists in the underlying constraint network. In other words, each hyper-arc of the constraint network will be labeled by a single *total constraint* [44]. We recall that a total constraint between a set S of variables is the intersection of all the mathematical constraints involving S.

4.2 Constraint handling using 2^k-trees

2^k-trees are hierarchical data structures based on the principle of recursive decomposition of space and are widely used in the domains of computer vision, robotics, computer graphics, image processing, pattern recognition systems and geographic information systems [126, 119, 120, 121, 144]. These structures are of interest mainly because of their ability to focus on aggregated subsets of data — rather than on the individual units of the aggregated entities. This focusing renders them particularly useful for performing *set operations*, often required for processing images (e.g., for reconstituting 3D images from binary projections or superimposing images).

There are a number of different ways of representing 2^k-trees. Since the complete pointer-based representation, including all types of nodes — black, white and gray — often require a considerable amount of overhead, the operations we present in this chapter are based on a linear representation of the trees. A linear 2^k-tree is the collection of leaf nodes, generally stored as a simple list. In our case, a linear 2^k-tree will contain the collection of feasible nodes, stored in a tree structure so that search and retrieval operations can be speeded up.

Linear 2^k-trees - elementary notions A 2^k-tree node represents a solution region $(I_1 \times I_2 \ldots \times I_k)$, where I_i is an interval along the i axis. A tree node is encoded using a succession of digits representing an integer in base 2^k. Each successive digit represents the quadrant subdivision from which the node originates. When two 2^k-tree nodes have the same representation except for the last digit, they are *condensed* (i.e., eliminated from the list of nodes and replaced with a code of $(n-1)$ digits followed by a marker, here denoted by *). Figure 4.1 gives an example of the quaternary linear encoding of a quadtree. In our case, the set of digits representing the nodes of a linear tree are stored in a tree structure so that node retrieval is in $O(log_{2^k})$ for a 2^k-tree (internal representation in Figure 4.1).

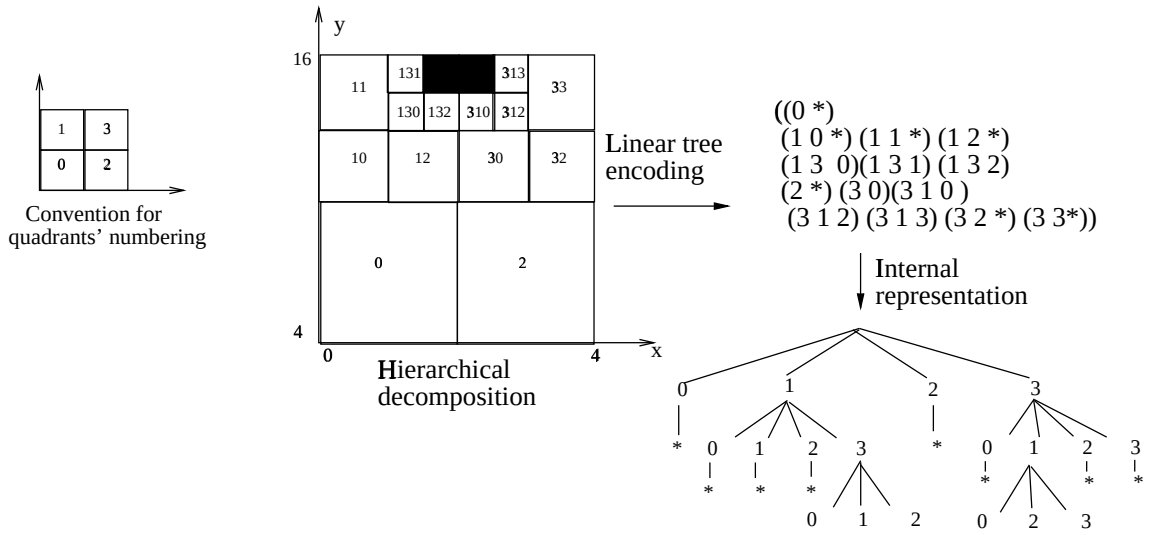


Figure 4.1: *Linear representation of a quadtree.*

In the worst case, the number of nodes in a linear 2^k-tree is $2^{k(l-1)}$ where l is the number of recursive quarterings. This case corresponds to the situation where each node of the level $\leq l-1$ is completely expanded and each node of the l^{th} level has (2^k-1) children, so that condensation cannot aggregate any set of nodes.

When searching for a node in a linear 2^k-tree, three cases can occur:

- the node is infeasible, and the search procedure reports failure
- the node is feasible, and it will be returned by the search procedure
- the node is partially feasible, and the search procedures returns the feasible sub-nodes that constitute it

Example : {*Tree-example is the quadtree in* Figure 4.1}
Retrieve$((313), \textit{Tree-example})) \rightarrow ((313))$;
Retrieve$((01*), \textit{Tree-example})) \rightarrow ((01*))$;
Retrieve$((133), \textit{Tree-example})) \rightarrow$ **false**;
Retrieve$((13*), \textit{Tree-example})) \rightarrow ((130)(131)(132))$;

The encoding procedure takes as input the dimensions of a node (list of intervals) and returns its digital encoding. Conversely, the decoding procedure returns the dimensions corresponding to the digital encoding of a node.

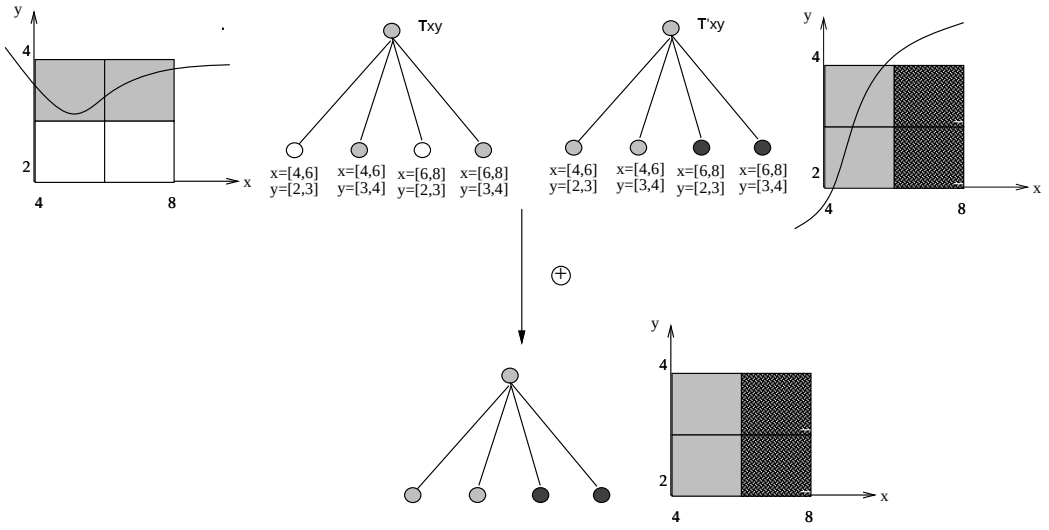


Figure 4.2: *Quadtree's intersection.*

Example :
$decode((313), ([0..4], [4..16])) \rightarrow ([5/2..3], [29/2..16])$;
$encode(([5/2..3], [29/2..16]), ([0..4], [4..16])) \rightarrow (313)$;

The *encode* , *decode* and *Retrieve* procedures can be carried out in $O(l)$ time complexity, where l is the number of recursive quartering (i.e., number of levels in the 2^k-tree). For further details concerning these procedures, we refer the reader to [126, 74].

The intersection, composition and projection operations on 2^k-tree we are interested in for handling numerical constraints are the set-theoretic operations which are most extensively discussed in computer vision, computer graphics and image processing. In this work, we consider the ordering $white < gray < black$ between colors and define the afore mentioned operators as follows:

Intersection: this operator — presented informally in section 3.3 — has, as input, the 2^k-tree representation of two relations R_1 and R_2, between the same variables $x_1, \ldots x_k$. The output is the 2^k-tree describing the solution space for the relation $R = \{R_1 \oplus R_2\}$ between variables $x_1, \ldots, x_k$. The rule for determining the feasibility of a node obtained by intersection can be expressed as follows:

$$color(node_1 \oplus node_2) = Max(color(node_1), color(node_2)) \tag{4.1}$$

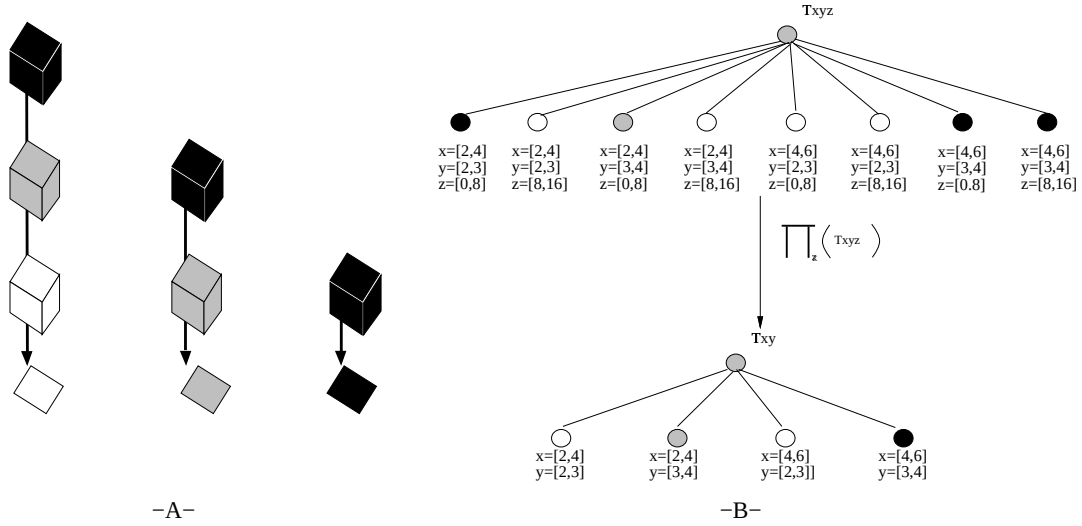


Figure 4.3: ***left(-A-)**: projection rule, **right (-B-)**: example of octree projection*

Intersection is performed between pairs of nodes which occupy the same region in space (see Figure 4.2).

The intersection algorithm traverses the two trees in parallel and examines corresponding nodes while it constructs the resulting tree [52]. When the linear tree encoding is used, the time complexity of this operator is in $O(N_1 + N_2)$ where N_1 and N_2 represent respectively the number of nodes in each tree.

Projection: this unary operator has, as input, the 2^k-tree encoding of a k-dimensional relation R_i between variables $x_1, \ldots, x_k$. The output is the $2^{(k-1)}$-tree describing the projection of the solution space of R_i over a hyper-plane $x_1 \ldots \times x_{i-1} \times x_{i+1} \times \ldots \times x_k$.

The projection algorithm simply traverses the tree and constructs a projection for each node of the original tree. When the projection of several nodes yields the same region in the $(k-1)$-dimensional space, the rule for determining the feasibility of the node obtained by projection can be expressed as follows (see Figure 4.3):

$$color(\prod_{x}(node_1)) = Min(color(node_i)) \tag{4.2}$$

where $node_i$ are the nodes having $node_1$ as facet and x is the variable removed by projection. Using linear trees, the time complexity of the projection operator is in $O(N)$, where N is the number of nodes in the tree to be projected.

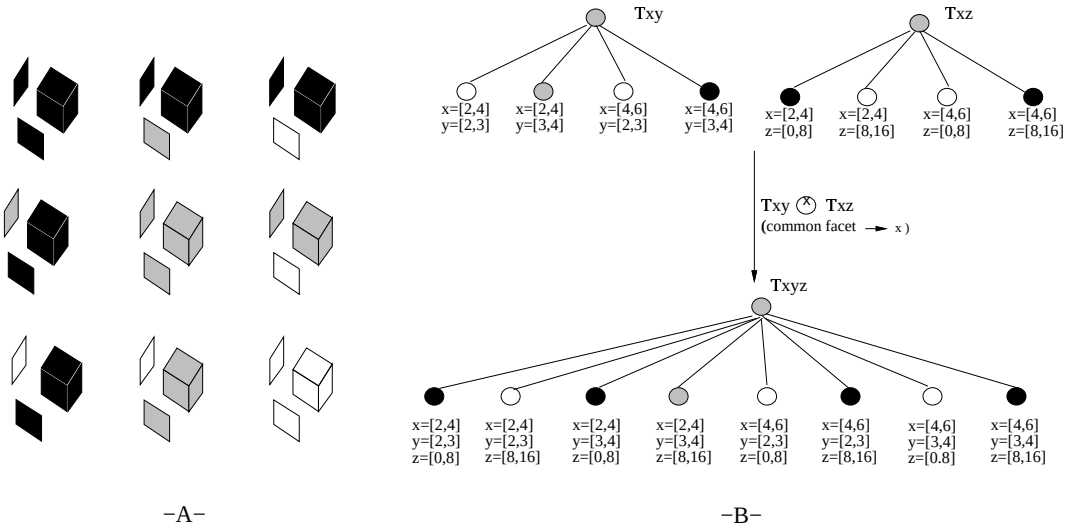


Figure 4.4: ***left(-A-)**: composition rule, **right (-B-)**: example of quadtrees composition.*

Composition: this binary operator has, as input, the 2^k-tree encoding of two relations $R_1(V_1)$ and $R_2(V_2)$, involving respectively the sets of variables V_1 and V_2 with $V_1 \bigcap V_2 \neq \emptyset$. The output is the 2^k-tree describing the solution space for the relation $R = \{R_1 \otimes R_2\}$ between variables $V_1 \bigcup V_2$.

Composition is performed between the nodes for which the common facet occupies the same region in space (see Figure 4.4, -B-). Each node n of the first tree can theoretically be composed with 2^{lc} nodes of the second one, where l is the number of digits encoding n and where $c = 2^{|V_1 \bigcup V_2| - Min(|V_1|,|V_2|)}$. For each node, $node_1$, of the first tree the composition algorithm searches for the nodes it can be composed with in the second tree. For each retrieved node, $node_2$, the following composition rule is then applied for determining the color of the resulting node (see Figure 4.4, -A-):

$$color(node_1 \otimes node_2) = Max(color(node_1), color(node_2)) \tag{4.3}$$

Using linear trees, the time complexity of the composition operator is $O(2^{l.c} N_1 log_{2^{|V_2|}} N_2)$ where l is the number of levels in the 2^k-tree decomposition, c is $2^{|V_1 \bigcup V_2| - Min(|V_1|,|V_2|)}$ and N_1, N_2 are respectively the number of nodes in the first and second linear tree.

Using the operators defined on 2^k-trees, *relaxation* as described before, can be implemented by first extending k-dimensional constraints into $k+1$-dimensional space by composition and then projecting back the result into k dimensions, as shown in Figure 4.5.

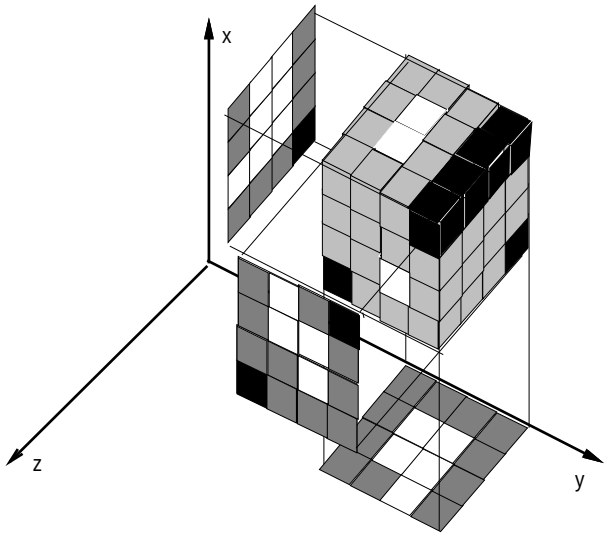


Figure 4.5: *Information on a 3-dimensional node can be simply derived by composing its facets (2-dimensional nodes), and vice versa, information on a 2-dimensional node can be obtained by projecting the 3-dimensional node over one of its facets.*

In the following section, we devise several consistency algorithms using the 2^k-tree representation of constraints.

4.3 Arc- and path-consistency algorithms

The constraint representation (2^k-trees) along with composition, projection and intersection, are the bases for applying consistency algorithms such as AC-3 [90] (for arc-consistency), PC-1 [99] and PC-2 [90] (for path-consistency) and their generalizations (for higher degrees of consistency).

We start by defining the notion of support used in the following sections.

Definition 4.2 *The support of a 2^k-tree node, n, is the set of nodes in a higher dimensional tree having n as facet.*

4.3.1 Arc-consistency

$\mathcal{T}$-AC-3 (Figure 4.6), which is the 2^k-tree based arc-consistency algorithm we propose, is a variant of AC-3 where the relaxation step is implemented using intersection, projection and composition of 2^k-trees.

Given a CCSP, (V, C, D), $\mathcal{T}$-AC-3 takes as input the set of 2^k-trees representing the

$\mathcal{T}$-**AC-3**($Trees$)

```
1  foreach T ∈ Tree do
2    foreach i ∈ Vars(T)
3      do Q ← Q ∪{(T, i)} od
4  od
5  while Q ≠ ∅ do
6        T(x_1, ..., x_k, i) ← Pop(Q);
7        T'(i) ← T(i) ⊕ ∏_(x_1,...,x_k)(T(x_1, ..., x_k, i))
8        if T'(i) ≠ T(i) then T(i) ← T;
9                              Q ← Q ∪(T, j)||i, j ∈ Vars(T)
10       fi
11 od
```

Figure 4.6: *Algorithm $\mathcal{T}$-AC-3.*

relations of C and returns for each variable of V, a unary tree that is arc-consistent. In the algorithm of Figure 4.6, $Vars(T)$ returns the set of variables associated with a given tree T.

Each relaxation step (line 7 in the algorithm of Figure 4.6):

$$T'(i) = T(i) \oplus \prod_{(x_1, x_2 \ldots, x_k)} (T(x_1, x_2 \ldots, x_k, i)) \tag{4.4}$$

removes from the domain of variable i, represented by a unary tree $T(i)$, the nodes that have no support in a higher dimensional tree $T(x_1, x_2 \ldots, x_k, i)$. According to the definitions of the operators $\oplus$ and $\prod$, the relaxation operation described by Equation (4.4) is monotonic: any node n' in $T'(i)$ is such that `color(n') > color(n)`, where n is the equivalent of n' in $T(i)$. In other words, a feasible node in $T(i)$ can be rendered infeasible (or partially infeasible) in $T'(i)$ but the converse is not true. Since the number of relations to revise, as well as the number of removable nodes is finite, proving the termination and correctness of $\mathcal{T}$-AC-3 can be done in a similar manner to that of AC-3 [90, 92]. The worst case running time of $\mathcal{T}$-AC-3 occurs when each revision step removes only one node from the unary tree $T(i)$. Let W be the number of nodes in the 2^k-tree of maximal size, each unary relation might be updated $O(2^{d-1}.W)$ time where d is the granularity of the decomposition[1]. Moreover, each modification in a relation adds to QA at most $O(e)$ new

[1] a unary linear tree can have at most 2^{d-1} nodes and projection is $O(N)$, where N is the number of nodes in a 2^k-tree.

*Iterative-$\mathcal{T}$-***AC-3**$(Trees, \varepsilon)$
1 Trees'←Trees
2 $T \leftarrow \{$*unary trees of* Trees$\}$;
3 **repeat**
4 $\overline{T' \leftarrow \mathcal{T}\text{-}AC\text{-}3(\text{Trees}')}$;
5 $precision \leftarrow Max_i(Interval(T(i)) - Interval(T'(i)))$;
6 $T \leftarrow T'$
7 Trees'← $Refine(\text{Trees}', T')$
8 **until** $precision \leq \varepsilon$

Figure 4.7: *Iterative $\mathcal{T}$-AC-3*

relations to revise, where e is the number of relations in the problem. Consequently, for a given granularity, we can state the following result:

Theorem 4.1 *$\mathcal{T}$-AC-3 computes an arc-consistent network representation of a CCSP, (V, C, D), in $O(W.e^2)$ where W is the number of nodes in the 2^k-tree of maximal size and e, the cardinality of C.*

Iterative arc-consistency

Fixing the granularity of the 2^k-trees decomposition for a problem prior to enforcing arc-consistency, can lead either to:

- unsatisfactorily rough approximations, when the level of precision is insufficient,
- a waste of effort, when the decomposition is carried out beforehand to the maximal possible precision,

In effect, in cases where large portions of the domains are not arc-consistent, they can be discovered and pruned more rapidly using rough decompositions. However, the results produced must be refined to avoid strong overestimations. 2^k-trees, as hierarchical structures, are particularly well adapted to dynamic refinements since improving their precision simply amounts to performing further subdivision of their gray nodes.

Starting with a rough approximation of the solution space, the algorithm of Figure 4.7 uses arc-consistency along with dynamic refinement of 2^k-trees, to determine iteratively sharper and sharper arc-consistent solutions. Arc-consistency determines in which region

the decomposition should be refined. Once the solution space description is refined, it is then checked for consistency in turn, in the more finely decomposed domain, until the precision required is reached. *Interval* $T(i)$ (step 5 of the algorithm) returns the interval projection of the unary tree $T(i)$.

It is important to notice that *Iterative-*$\mathcal{T}$-AC-3 requires *conservative* approximations as input. This means that the linear trees of the input must include both gray and white nodes in general. Additional markers are then used to differentiate partially feasible from completely feasible nodes. In case a non-conservative representation is used (white nodes only), the coarse approximation used in the early stages of the algorithm may lead to removing improperly large portions of feasible regions.

The refinement step of *Iterative-*$\mathcal{T}$-AC-3 (line 7 of the algorithm) consists of removing from each linear tree the nodes falling outside the refined domains. The remaining gray nodes are then further subdivided to produce a sharper approximation of the solution region represented by the tree:

Procedure*Refine*$(Trees, Unary - trees)$:
foreach $T \in Trees$ **do**
  $V \leftarrow Vars(T)$;
  foreach $i \in V$ **do**
    $T \leftarrow T \oplus Unary - trees(i)$
    foreach *gray node*$, n \in T$ **do**
      $subdivide(n, T)$, **od**

Once a sufficiently precise solution has been identified using *Iterative-*$\mathcal{T}$-AC-3, a non conservative approximation can be obtained by removing the remaining gray nodes.

We describe in chapter 6 the successful use of *iterative-*$\mathcal{T}$-AC-3 for approximating the solutions of one of the toughest benchmark of a recent report on interval methods, also resolved using the `Newton` arc-consistency algorithm [7].

4.3.2 Path-consistency

We present in this section a path-consistency algorithm for binary constraints. A k-consistency algorithm is presented later that provides a more adapted handling of problems with arbitrary arities.

$\mathcal{T}$-**PC-2**($Trees$)
1 $Q \leftarrow \{(i,j,k) \| (i<j) and (k \neq i,j)\}$
2 **while** $Q \neq \emptyset$ **do**
3 *Select and delete a* $path(i,k,j) from Q$; ,
4 $T'(i,j) \leftarrow T(i,j) \oplus \prod_k (T(i,k) \otimes T(k,j))$;
5 **if** $T' \neq T$ **then** $Q \leftarrow Q \cup related\text{-}paths((i,k,j))$;

Figure 4.8: *Algorithm PC-2*

Analogously to the case of arc-consistency, we present $\mathcal{T}$-PC-2, a variant of PC-2 adapted to the 2^k-tree representation of constraints. The procedure *related-paths* used by the algorithm, returns the set of length two paths that need to be revised if the revision of a path (i,k,j) has led to the removal of some nodes in the quadtree $T(i,j)$.

The relaxation operation required by $\mathcal{T}$-PC-2:

$$T'(i,j) = T(i,j) \oplus \prod_k (T(i,k) \otimes T(k,j)) \tag{4.5}$$

is implemented using composition($\otimes$), intersection($\oplus$) and projection($\prod$) of quadtrees.

Similarly to the case of arc-consistency, and according to the definitions of $\oplus$ and $\otimes$ for 2^k-trees, the relaxation operation described by Equation (4.5) is monotonic: any node n' in $T'(i,j)$ is such that `color(n') > color(n)` where n is the equivalent of n' in $T(i,j)$. Since the 2^k-tree decomposition discretizes the solution space, showing that PC-1 (and hence PC-2) terminates and computes a path-consistent network using the relaxation operation $T'(i,j) = T(i,j) \oplus T(i,k) \otimes T(k,j)$, can be done similarly to the case of discrete-domains CSPs (see [99]). Path-consistency for a CCSP can therefore be achieved by PC-2 using the relaxation operation (Equation (4.5)) by way of revision step. The worst case running time of PC-2 occurs when each revision step suppresses only one node from the considered relation (i.e., the node becomes black). Let W be the number of nodes in the quadtree of maximal size, each relation might be updated $O(W)$ time. Each revision step requires composing two *binary* relations. For a fixed precision, the complexity of this composition is $O(W log_4(W))$ (see section 4.2 for the complexity of quadtrees composition). Moreover, the number of relations in the constraint network is in $O(n^2)$ and each modification in a relation adds $O(n)$ new paths to revise. Consequently:

Theorem 4.2 *PC-2 computes a path consistent network representation of a binary CCSP, (V, C, D), in $O(n^3.W^2 log_4(W))$ where W is the number of nodes in the quadtree of maximal size and n is the cardinality of V.*

In chapter 6 we give an example of non-linear problems solved using $\mathcal{T}$-PC-2.

4.4 Algorithms for higher degrees of consistency

Propagation algorithms can be improved by taking into account the following observations:

- each relaxation step is likely to affect only subparts of the revised 2^k-tree (only some subtrees are modified),
- if a node is modified in a given tree, it is straightforward to determine which nodes will be modified in the related higher-dimensional and lower-dimensional trees,

For example, in the case of path-consistency, the relaxation step of Equation (4.5) requires to recompute completely the composition of $T(i,j)$ and $T(i,k)$ which is unnecessary in the most general case. In effect, only some nodes of the composition $T(i,j) \otimes T(i,k)$ are affected by a modification in $T(i,j)$, $T(i,k)$ or $T(j,k)$. The idea is that a modification in one of those trees can be propagated selectively to the relevant sub-trees of $T(i,j) \otimes T(i,k)$. Conversely, any modification in $T(i,j) \otimes T(i,k)$ can in turn be propagated selectively to the involved subparts of the related lower-dimensional trees. This would spare unnecessary operations on the unchanged nodes of the trees.

Based on these considerations, we propose a variant of Cooper's k-consistency algorithm [27], adapted to the 2^k-tree representation of constraints. Cooper's algorithm is a variant of Freuder synthesis algorithm [46].

4.4.1 Retrieving the solution regions to remove

We start by devising two simple algorithms, *Expand-node* and *Project-node*, needed for implementing the 2^k-tree version of Cooper's algorithm.

Both algorithms use the traditional procedures for encoding decoding and retrieving nodes in 2^k-trees, as described in section 4.2.

Expand-nodes$(node, Vars_i, var', Int', Dimensions)$:

```
1  N ← ∅;
2  s ← Length(I_{i+1})/width(n);
3  for b := Min(I_{i+1}) to Max(I_{i+1})
4      do
5      Dims ← Append(Decode(n, Dimensions), [b..b + s]);
6      n' ← encode(Dims, Dimensions);
7      if Retrieve(n, Tree(Vars_i ⋃{x_{i+1}}));
8        then
9              N ← N ⋃{n'};
10     fi
11 od
12 return N
```

Figure 4.9: Algorithm Expand-node.

Retrieving higher-dimensional nodes to remove: The algorithm `Expand-node` takes as input:

- a node n of dimension i representing a solution region for i variables $x_1, \ldots x_i$,
- a variable x_{i+1},
- and the current domain (interval), I_{i+1}, for variable x_{i+1}

The output is the list of nodes having n as facet within interval I_i in the $2^{(i+1)}$-tree involving the variables $\{x_1, \ldots x_{i+1}\}$.

Once a node n has been blacked out during a constraint propagation process, the procedure *Expand-node* can be used to propagate the corresponding illegal labeling to higher-dimensional trees.

In the algorithm of Figure 4.9, the following notations and conventions are adopted: *node* is a i-dimensional node representing a solution region for the variables $\{x_1, \ldots, x_i\}$, with $x_i \neq x_{i+1}$. $Vars_i$ denotes the set $\{x_1, \ldots, x_i\}$, I_{i+1} is the current interval for x_{i+1} and *Dimensions* is the set of original intervals for the variables $\{x_1, \ldots x_i, x_{i+1}\}$. $width(node)$ returns the size of *node*, given by its number of digits. $Length(I_{i+1})$ returns the interval length of I_{i+1} and $Tree(Vars)$ returns the 2^k-tree associated with the variables $Vars$.

The algorithm collects in N the nodes of the tree involving $x_1 \ldots x_i, x_{i+1}$ that have *node* as facet and such that x_{i+1} takes its values within I_{i+1}. The main loop (steps 3-11)

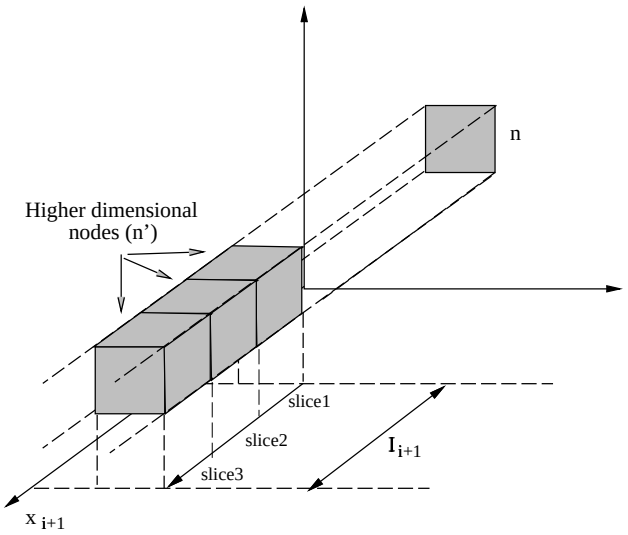


Figure 4.10: *Determining higher dimensional node*

iterates on the successive slices of the interval I_{i+1} along the x_{i+1} axis. A slice has a length proportional to the size of *node* (see Figure 4.10).

Retrieving lower-dimensional nodes to remove: The algorithm `Project-node` takes as input:

- a node n of dimension i, representing a solution region for i variables, $x_1, \ldots x_i$,
- a variable x_k such that $x_k \in \{x_1 \ldots x_i\}$,
- and the current interval, I_k, for x_k

If n has no neighbor within interval I_k along the x_k axis in the 2^i-tree involving the variables $\{x_1, \ldots x_i\}$, this procedure returns the projection of n, else it reports failure.

Provided that n has been identified as illegal during a constraint propagation process, the projection of the node n over the axis x_k can be blacked out in turn only if n has no legal neighbor along the x_k axis (i.e., only if the projection has no alternative legal support) (see Figure 4.12). *Project-node* checks for the existence of alternative legal supports. In case no support exists, it returns the projection of node n, as node to remove from the tree involving variables $\{x_1 \ldots x_i\} - \{x_k\}$.

In the algorithm of Figure 4.11, the notations and conventions are similar to those employed for the algorithm *Expand-node*. The call *Domain*(x_k) in step 2, returns the

Project-node$(n, Vars_i, x_k, I_k, Dimensions)$:

```
1  new-node ← ∏_{x_k}(node);
2  new-dims ← Dimensions − Domain(x_k);
3  N ← ∅
4  s ← Length(I_k)/width(n);
5  for b := Min(I_k) to Max(I_k)
6      do
7      Dims ← Append(Decode(new-node, new-dims), [b..b + s]);
8      n' ← encode(Dims, Dimensions);
9      if n' ≠ n
10       then
11             if Retrieve(n', Tree(Vars_i))
12               then N ← N ∪{∏_{x_k}(n')};
13             fi
14     fi
15 od
16 foreach n' ∈ N
17   do new-node ← new-node ⊕ n';
18 od
19 Return(new-node);
```

Figure 4.11: Algorithm Project-node.

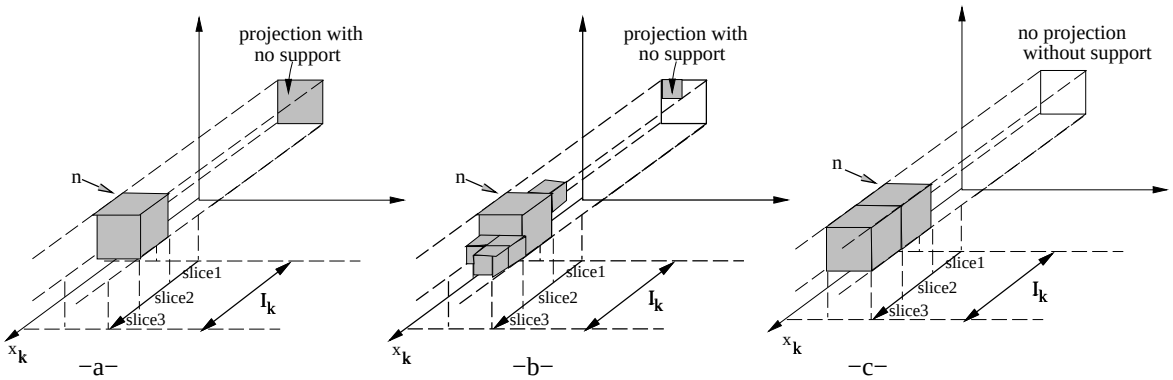


Figure 4.12: *Determining whether a node n still have legal supports in a higher-dimensional tree.*

domain of the variable x_k. Similarly to the case of the algorithm *Expand-node*, the main loop (steps 5-15) iterates on the successive slices of interval I_k, collecting in N the neighbors of n within I_k along the x_k axis. The existence of such a neighbor means that the projection of n still has some supporting nodes. The following cases must be considered (see Figure 4.12):

- no supporting neighbor exists: in this case the projection of n can be removed from the $2^{(i-1)}$-tree involving the variables $Vars_i - \{x_k\}$ (Figure 4.12 -a-),
- some supporting neighbors, smaller than n, exist: in this case some regions of the projection may have supports while other not (Figure 4.12 -b-). The sub-regions of the projection that can be removed are those having no support. These regions are given by the intersection of the projection of n and the projections of the supports (step 17 of the algorithm),
- a supporting neighbor exists that has the same size as n: in this case the whole projected region still has an alternative support and consequently, it cannot be removed from the $2^{(i-1)}$-tree involving the variables $Vars_i - \{x_k\}$ (Figure 4.12 -c-).

4.4.2 A variant of Cooper's algorithms

This section describes the algorithm $\mathcal{T}$-KC, a variant of Cooper's algorithm based on the 2^k-tree representation of constraints. Given a CCSP, (V, C, D), $\mathcal{T}$-KC constructs a linear tree, $Tree(Vars_i)$, for each cardinality-i subset $Vars_i$ of V. It takes as input, the set of complete[2] 2^k-trees representing the constraints of C. *Full-tree*$(Vars_i)$ returns the complete 2^k-tree associated with the variables $Vars_i$. Each linear tree, $Tree(Vars_i)$, is initialized with the white nodes of the corresponding complete 2^k-tree. In case no explicit complete 2^k-tree exists for a subset of variables $Vars$, $Tree(Vars)$ is initialized to a condensed linear tree ((*)), which mean that all possible values are considered as legal, a priori. In the following, $\mathcal{T}$ will denote the set of linear trees constructed.

Intervals(var_i) contains the legal intervals for a given variable var_i. This list is initialized with the domain of var_i and is updated when higher-dimensional nodes are propagated to unary linear trees (step 43).

[2]including white, gray and black nodes.

List is a structure containing the illegal nodes to propagate in $\mathcal{T}$, until total relaxation is reached (i.e., no more propagation is possible). *List* is initialized using the illegal nodes (black nodes) of the complete trees.

Step 2a propagates the illegal nodes to supersets of $Vars_i$ of cardinality $i+1$. The determination of which node removing from the cardinality-(i+1) linear trees is done using the procedure *Expand-node*.

Similarly, Step 2b propagates the illegal nodes to subsets of $Vars_i$ of cardinality $(i-1)$ and the determination of which node removing from the cardinality-(i-1) linear trees is done using the procedure *Project-node*. The information stored in the structures *Intervals* allows for searching higher- and lower-dimensional nodes to remove only within the legal regions —instead of the whole original domain.

In addition, the algorithm uses the following utilities:

- $Domain(var_i)$ (line 3): returns the domain for the variable var_i,
- $Dimensions(Vars_i)$ (line 25): returns the respective domains of the variables $Vars_i$,
- $Remove\text{-}node(node, Tree(Vars_i))$: removes a node from the linear tree involving the variables $Vars_i$. The removal of a node in a n node 2^k-tree (including condensation) has a time complexity of $O(l.n)$ where l is the level of decomposition (number of recursive quarterings).

$\mathcal{T}-KC$:

```
1  {Step1 : initialization}
2  for i := 1 to n do
3      Interval(var_i) ← Domain(var_i);
4      for i := 1 to k do
5          foreach i-tuple Vars_i of variables do
6              if Full-Tree(Vars_i) then
7                                          begin
8                                              Tree(Vars_i) ← white-nodes(Full-Tree);
9                                              foreach node ∈ black-nodes(Full-Tree) do
10                                                 Append(List, (node, Vars_i, i));
11                                             od
```

```
12                                  end
13                              else Tree(Vars_i) ←' ((*)) fi od od
14      {Step2}
15      while List ≠ ∅ do
16              (node, Var_i, i) ← Pop(List);
17              {2a} if i ≤ k
18                      then {Propagate to level i + 1}
19                              foreach var' ∉ Vars_i do
20                                  Int' ← Interval(var');
21                                  Vars_{i+1} ← Vars_i ⋃{var'};
22                                  Dim' ← Dimensions(Vars_{i+1});
23                                  N ← Expand-node(node, Vars_i, var', Int', Dim');
24                                  foreach n ∈ N do
25                                    then Append(List, (n, Vars_{i+1}, i + 1));
26                                         Remove-node(n, Tree(Vars_{i+1})); od od fi
27                      {2b} if i > 1
28                              then {Propagate to level i − 1}
29                                      for j := 1 to i do
30                                          Vars_{i−1} ← Vars_i − {var_j};
31                                          I_j ← Interval(var_j);
32                                          Dim' ← Dimensions(Vars_i);
33                                          N ← Project-node(node, Vars_i, var_j, I_j, Dim');
34                                          foreach n ∈ N do
35                                            Append(List, (n, Vars_i, i));
36                                            Remove-node(n, Tree(Vars_i));
38                                          od
39                                      od
40                                      if i − 1 = 1
41                                        then
43                                              Interval(var_j) ← Interval(var_j) − node
44                                      fi
45                      fi
```

Unlike Freuder's algorithms, in which a relation of arity i is relaxed only after all relations with smaller arities have been completely relaxed, Cooper's algorithm does not fix an order in which illegal labeling must be propagated. The same principle is kept in our algorithm since it enables an adaptive choice of which region to remove first, thus providing more flexibility for specific optimizations. However, the notion of counters — one of the major improvements proposed by Cooper in his algorithm — is abandoned, for the following reasons:

- 2^k-trees might have a large number of minimal size nodes and the space overhead engendered by an explicit maintenance of counters is likely to degrade significantly the performances of the algorithm. In effect, keeping the principle of Cooper's algorithm for handling supports would imply the introduction of an individual counter for each pair of *(minimal size node, variable)*. This loses the advantage provided by a hierarchical decomposition which avoids the explicit handling of individual values,

- a node n of dimension i is removed when there exists a relation with arity i+1 having no projections over i equal to n. Counters have been introduced in order to avoid a useless re-computation of each possible projection, including those having no relation with the label to remove. In our case, we can benefit from the ordered nature of continuous solutions spaces along with the locational relations existing between labels (regions to remove) to compute and retrieve easily the relevant sub-regions to be projected (see the algorithms *Expand-node* and *Project-node*).

The proof of correctness for our algorithm is similar to the one proposed by Cooper [27]. The set $\mathcal{T} = \{Tree(vars_i) : Vars_i \subseteq V, \|Vars_i\| = i, 1 \leq i \leq k\}$ is defined as a k-solution if it satisfies the following properties:

1. $Tree(Vars_i) \subseteq$ *white-nodes*$(Full\text{-}tree(Vars_i))\forall Vars_i \subseteq V : \|Vars_i\| = i, 1 \leq i \leq k$.

2. $node \in Tree(Vars_i) \wedge Vars_i \subseteq Vars_k \Rightarrow \exists node' \in Tree(Vars_k)$ such that *node* is a facet of *node'*.

3. There does not exist a set $\mathcal{S}$ of linear trees satisfying properties (1) and (2) and such that $\exists\ Tree_1 \in \mathcal{T}, Tree_2 \in \mathcal{S}$ with $Tree_1 \subseteq Tree_2$ (i.e., the set of feasible nodes of $Tree_1$ is included in the set of feasible nodes of $Tree_2$).

The set of linear trees, $\mathcal{T}_{fin}$, obtained when the algorithm terminates, is then proved to satisfy properties (1), (2) and (3) as follows:

- $\mathcal{T}_{fin}$ satisfies property (1) because $\mathcal{T}$ is initialized with the white nodes of the complete 2^k-trees representing C and because no new node is added during the propagation of illegal labeling.
- $\mathcal{T}_{fin}$ satisfies property (2) because each node in $Tree(vars_i)$ is a facet of a node in $Tree(vars_i \bigcup var')$(with $var' \notin Vars_i$) and each node in $Tree(vars_i - \{var'\})$ is a facet of a node in $Tree(vars_i)$. Applying these two properties inductively yields property (2).
- $\mathcal{T}_{fin}$ satisfies property (3) because any set of linear trees satisfying the constraints C and verifying the properties (1) and (2) must be totally relaxed. Since relaxation is the only way labelings are eliminated from the trees $T(vars_i)$, each tree $T'(vars_i)$ must satisfy $T'(vars_i) \subseteq T_{fin}(vars_i)$.

Hence, the following result can be stated:

Theorem 4.3 *$\mathcal{T}$-KC computes a k-consistent network representation of a CCSP, (V, C, D).*

Complexity The inner loop of the initialization step (steps 4-13) is executed at most $\Sigma_{i=1}^{k} C_n^i$ where C_n^i is the number of order-i relations — we assume that black and white nodes are directly stored separately when the complete 2^k-trees are constructed. In the worst case, the number of black nodes in a 2^i-tree is $2^{i(l-1)}$ where l is the number of recursive quarterings. This case corresponds to the situation where each node of the level $\leq l-1$ is completely expanded and each node of the l^{th} level has $(2^i - 1)$ children, so that condensation cannot aggregate any set of nodes. The loops in *Step 2a* and *Step 2b* are executed at most $\Sigma_{i=2}^{k} i.C_n^i 2^{i(l-1)}$ times. Hence the worst case time complexity is $O(\Sigma_{i=1}^{k} C_n^i 2^{i(l-1)})$.

4.5 Related work

Arc-consistency techniques In [93], Mackworth *et al.* propose a hierarchical arc-consistency algorithm, HAC, that exploits the internal structuring of domain values into

hierarchies of subdomains, for the efficient treatment of constraint satisfaction problems with large domains. The arc-consistency algorithm we propose, $\mathcal{T}$-AC-3, can be seen as a particular instance of HAC where the abstract structuring of the domains is given by the 2^k-trees representation and where the hierarchical predicates required for domains revision are implicitely implemented using the logical operators on 2^k-trees. Similarly, *iterative*-$\mathcal{T}$-AC-3 can be seen as an extension of HAC, where the abstract structuring of the domain is itself refined iteratively.

From the perspective of continuous domains, the arc-consistency techniques presented in this chapter mainly differ from their purely numerical counterparts by the fact that: *i.* the unary projections are not computed using interval arithmetic methods, *ii.* the risk of infinite looping is avoided using total constraints and by explicitly handling individual solution regions, and *iii.* multiple solutions (above a certain degree of precision) are systematically searched for. One important point to observe is that, since 2^k-trees provide a compact description of the solution spaces and preserve the locational relations between the solution regions, the complexity of the 2^k-tree based consistency techniques does not depend on the existence of multiple solutions. In effect, multiple solutions only translate into the existence of topologically disconnected nodes in the 2^k-tree. Finally, since using 2^k-trees limits the overestimation to a fixed precision — contrary to basic interval arithmetic operations for deriving the ranges of functions which often results in strong overestimates— the arc-consistency algorithms we propose generally provide better results than the traditional Waltz arc-consistency implementations and their derivates. This is related to the fact that in interval arithmetic, as in this example of subdistributivity, $a(a-1) \subseteq a - a^2$, determining the best order of operations will affect the quality of results (i.e. best enclosures). This task is complex and generally not performed in the traditional implementations of Waltz's algorithm, which often leads to strong overestimations of enclosures. Using 2^k-trees, this problem is avoided since the distributivity property is maintained. For example, the representation of $y = a(1-a)$ is strictly identical to the representation of $y = a - a^2$.

While comparing our approach with more sophisticated techniques, such as Newton box-consistency or 2-B-consistency, several points have to be discussed.

In terms of generality and robustness, $\mathcal{T}$-AC-3 offers more guarantees than `Newton` and 2-B-consistency. As already mentioned, fixed-point iteration schemes are sensitive to

the existence of cycles, and the use of locally convergent technique by means of narrowing operators may pose convergence problems. For that matter, Newton imposes applicability conditions to guarantee satisfactory behaviors of the algorithm, and Lhomme [88] uses the notion of K-B(w)-consistency to characterize the partial solutions produced by the algorithms when they end abnormally. On the other hand, provided that proper initial conditions hold, Newton produces sharp approximations of the solution space outer bounds more rapidly — globally convergent operators (such as binary-search) being notoriously slower than locally convergent ones. In addition, by handling individual constraints of arbitrary arity as a whole, Newton reduces the *variables dependency problem* which is likely to be more important using our approach, where n-ary constraints are decomposed into ternary ones. However, when simultaneous constraints exist between a given set of variables, the use of total constraints will, on the contrary, reduce the variable dependency problem.

Note that the decomposition into ternary constraints is only motivated in our case by the objective of reducing the arity of 2^k-trees. In the 2-B-consistency approach, it obeys more demanding considerations related to the computation of interval functions using interval arithmetic.

Both Newton box-consistency and 2-B-consistency compute a convex enclosing approximation of the solution space with no concern about the identification of multiple solutions.

Note finally that since the construction of a 2^k-tree requires finding the zeros of individual equations, the techniques we propose are clearly not adapted to numerical problems involving single equations. The major advantage of 2^k-trees lies in the easy and reliable constraint *combinations* operators they provide and the approach discussed in this chapter targets exclusively the problems defined as sets of *several* constraints.

Path-consistency using 2^k-trees Tanimoto [132] has shown the interest of 2^k-trees for handling a particular type of continuous constraints (spatial constraints) and also proposed the implementation of PC-2 using this constraint representation. In Tanimoto's approach, quadtrees are used to represent spatial solution regions in 2D, instead of mathematical solution spaces derived from numerical equations. These quadtrees are derived from canonical spatial constraints using computer graphics techniques (plane-sweeping

algorithms [118]). Tanimoto's work and ours, though carried out independently with different objectives, lead to related methodologies. This can be seen as favorably illustrating the relevance of discretized hierarchical decomposition for designing reliable continuous consistency techniques.

***k*-consistency algorithms:** To the best of our knowledge, and except for the works addressing directly the issue of computing n-consistent solutions [75, 122], the work of Lhomme [88] is the only explicit proposal of algorithms for higher degrees of consistency than path-consistency in continuous domains. Lhomme defines the notion of K-B-consistency (see section 2.5.5), where the k-consistency checkings are restricted to the outer bounds of the solution space. Because of the convexity assumption it relies on, K-B-consistency is weaker than k-consistency. In effect, the spurious solutions existing between the k-consistent outer bounds are not removed from the labels of variables. Therefore the approach Lhomme proposes is not directly comparable to ours.

4.6 Discussion and summary

This chapter devised several consistency algorithms based on a 2^k-tree representation of constraints. In contrast to the existing numerical consistency techniques, our approach aims at working with the *feasibility space* delimited by constraints rather than variables' domains. By allowing the explicit handling of solution regions, 2^k-trees offer a simple means to maintain pointwise relation between variables, thus faciliting the implementation of algorithms for higher degrees of consistency than arc-consistency. We will conclude our discussion on this topic by summarizing the main arguments we see for handling numerical constraints using 2^k-trees:

1. *Discretization prevents infinite looping:* At each relaxation step performed by consistency algorithms on 2^k-trees, the intervals contained in the involved labels are constructed by an implicit binary search: each successive relaxation step refines the interval bounds to an interval half the size of the previous one until maximum granularity is reached. Consequently, the decomposition into 2^k-trees has the important advantage of ruling out infinite cycling of the propagation algorithm as observed for the Waltz algorithm applied to continuous domains. While the majority of consis-

tency algorithms perform unstable fixed point iterations, the binary search method using the 2^k-tree decomposition guarantees stability and convergence. The methods presented are consequently robust even for cyclic networks, explicit cycles being eliminated by the computation of total constraints (see section 3.3).

2. *Numerical tools are required only for solving individual equations:* the numerical handling of simultaneous set of constraints is replaced by logical operators on 2^k-trees. The only numerical tools needed are those for finding the zeros of individual binary and unary equations — used for determining the colors of nodes during the 2^k-trees construction.

3. *Distinguishing multiple k-consistent solutions does not increase the complexity* of 2^k-tree based consistency techniques: This is not the case using standard interval-based propagation techniques where each interval split generates a new sub-CSP to be checked for consistency separately.

4. *The precision of results can be improved deterministically:* results are refined by increasing the resolution parameter of the 2^k-tree decomposition and the accuracy does not depend upon convergence guarantees, as it is often the case when using standard numerical-based techniques.

These features provide a basis for designing reliable k-consistency algorithms, a question marginally addressed in the literature so far.

Chapter 5

Global Consistency for Continuous CSPs

Globally consistent networks allow efficient backtrack-free resolution schemes. While enforcing this property generally engenders prohibitive computational costs, it has been shown that particular conditions render it tractable for certain classes of problems. For binary discrete and temporal constraints, it is known that when constraints are *convex*, a low order of consistency (namely path-consistency) is sufficient to ensure global consistency. The contribution of this chapter is to show how, in continuous domains, this result can be generalized to ternary and in fact arbitrary n-ary constraints. This leads to polynomial time algorithms for computing the globally consistent labelings of a large class of continuous CSPs.

5.1 Convexity and global consistency

In a *globally consistent* network (also called *decomposable* or strongly n-consistent) each possible constraint, of arity less than or equal to the number of variables, is completely *explicit*. This means that any partial consistent instantiation of a subset of variables can be extended to a *solution* with no backtracking [35], a process which can generally be carried out in linear time.

Minimality, defined by Montanari [99] for binary networks, is a weaker property[1] guaranteeing that each pair of values allowed by the constraints participates in at least one solution.

[1] a decomposable network being always minimal while the opposite is not true

Much work has been done to determine under which conditions global consistency and minimality are tractable. According to van Beek's survey of related issues [137], the restrictions useful for enforcing minimality and decomposability using local consistency algorithms fall into one of the following categories:

1. *Restrictions on the types of constraints:* it has been shown for example, that relationships exist between the local level of consistency required for reaching global consistency and the size of the variable domains (Dechter [35]) or the tightness of constraints (van Beek & Dechter [138]). Other work has demonstrated the role played by characteristics of the constraints themselves. As an example, global consistency can be enforced using path-consistency algorithms when the constraints are monotone (Montanari [99]) or when they specify convex continuous intervals (Dechter *et al.* [38], van Beek [136]). In the same spirit, a condition sufficient for testing the satisfiability of a network is arc-consistency when the constraints are functional and monotone (Deville & Hentenryck [41]), and path-consistency when they are bijective (David [33]). In [137], van Beek generalizes the results of Montanari and Deville & Hentenryck by identifying an important property of discrete constraints called *row-convexity*.

2. *Restrictions on the constraint network topology:* the typical example of tree-structured networks illustrates this category of restrictions. For this particular case, minimal networks are known to be computable using path-consistency algorithms (Montanari [99]) and solutions can be derived backtrack-free after enforcing arc-consistency (Freuder [47]). Generalization of these results to the case of k-trees has been proposed by Freuder (Freuder [47]) who has also identified a relationship between a property of the network (called its width) and the degree of local consistency required for reaching global consistency. Heuristic methods based on neighborhood relationships between the nodes of the network have also been proposed by Dechter [37].

3. *Restrictions on both issues:* Dechter and Pearl [39] show that when constraint networks can be formulated as a causal theory, solutions can be derived backtrack-free.

Our work identifies useful restrictions on the topology of the feasibility spaces delimited

by constraints. It consequently falls within the class of works restricting the nature of constraints.

The results we present are directly inspired by the work of van Beek on row-convex discrete CSPs [137] and the work of Dechter *et al.* on simple temporal problems [38], both of which have shown the importance of convexity for enforcing minimality and decomposability using *local consistency algorithms*. Before going into more detail, we recall briefly the main definitions and results relative to simple temporal problems and row convex discrete problems.

Simple temporal problems A temporal constraint satisfaction problem (TCSP) is a particular CSP involving a set of variables representing time points. Each variable takes its values within continuous domains (intervals). It is known that solving general TCSPs is NP-hard [38]. In [38], Dechter, Meiri and Pearl identify particular instances of this problem (called STPs) admitting polynomial time solutions.

Definition 5.1 (Dechter, Meiri and Pearl) *A TCSP in which all the constraints specify a single interval is called a simple temporal problem (STP)*

In [38], Dechter, Meiri and Pearl show that determining the consistency of an STP and computing its minimal network can be done in $O(n^3)$ using the Floyd-Warshall's algorithm — which is shown to be equivalent to PC-1. It is also shown in the same work that solutions to a general TCSP can be computed by solving individually each subproblem corresponding to an STP and then combining the results. We will show later how similar results can be stated for general CCSPs.

Row convex CSPs van Beek uses the matrix-based representation of constraints proposed by Montanari [99]. A discrete binary relation R_{ij} is represented by a $(0,1)$ matrix with $|D_i|$ rows and $|D_j|$ columns where D_i and D_j are respectively the domains of the variables i and j. 1 entries represent legal value combinations and 0 entries represents illegal ones. A partial convexity property of matrices is then exhibited, which allows for polynomial time global consistency algorithms:

Definition 5.2 (van Beek) *A $(0,1)$ matrix is row convexif and only if in each row, all of the ones are consecutive.*

van Beek's results are demonstrated using the following Lemma — a generalization to discrete domains of the finite intersection property enjoyed by sets of convex continuous spaces:

Lemma 5.1 (van Beek) *Let $\mathcal{F}$ be a finite collection of $(0,1)$-row vectors that are row convex and of equal length, such that every pair of row vectors in $\mathcal{F}$ have a non-zero entry in common; that is, their intersection is not the vector with all zeros. Then all row vectors in $\mathcal{F}$ have a non-zero entry in common*

Although the convexity features exploited in the case of STPs are stronger than those specified by row-convexity — the fact that each variable takes its values within a single convex interval means that the solution space is a convex hyper rectangle — the results on both issues are based on the same mathematical argument, discussed in more details later. This analogy has motivated our research on a possible extension to more general classes of continuous problems.

The rest of this chapter is structured as follows: section 5.2 shows how the theoretical results and algorithms developed for STPs and row convex problems can be transferred to general CCSPs, using *convexity*. It also demonstrates that, contrary to the case of discrete problems, these results are tractably generalizable to n-ary constraints ($n > 2$) in continuous domains. A new concept of consistency called $(r, r-1)$-relational consistency is introduced that leads to polynomial time algorithms for computing globally consistent labeling for n-ary CCSPs. Section 5.3 shows that partial convexity features, weaker than full convexity, remain of interest for reaching global consistency tractably. Section 5.4 then presents how the useful convexity and partial convexity properties introduced earlier can be checked for reliably on the 2^k-tree approximation of constraints. A classification of general continuous CCSPs, with respect to their complexity predictability is proposed in section 5.5. Section 5.5.4 discusses the issue of general CCSPs — those verifying no particular convexity property. It proposes a decomposition approach that might be of interest either for bounding the complexity of practical CCSPs *a priori*, or for enabling search schemes within sets of convex sub-problems.

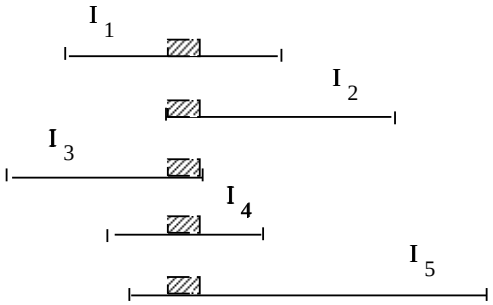


Figure 5.1: *Helly's Theorem in one dimension: if each interval is convex and if each pair of intervals has a non null intersection, then the whole set of intervals have a non null common intersection (shaded interval)*

5.2 Reaching global consistency using convexity

For both simple temporal problems and row-convex discrete problems, it has been observed that *convexity* of the constraint relations means that path-consistency is sufficient to ensure a globally consistent labeling. This result is proved using Helly's Theorem (illustrated in Figure 5.1), or its extension to discrete problems:

Theorem 5.1 (Helly) *Let F be a finite family of at least $n+1$ convex sets in R^n such that every $n+1$ sets in F have a point in common. Then all the sets have a point in common.*

Helly's Theorem can be applied to show that for each assignment of n variables $x_1, x_2, ..., x_n$, there exists a consistent value which can be assigned to x_{n+1} in the following way. Since the constraint network is binary, the only constraints existing between $x_1, ..., x_n$ and x_{n+1} are individual constraints between $x_i, i \in \{1.., n\}$ and x_{n+1}. Since the constraints are convex, every variable x_i already assigned constrains x_{n+1} to a single interval. Path-consistency ensures that every pair of such intervals intersect each other. Thus, by Helly's Theorem, there must exist a common intersection of all the intervals (i.e., at least one value for x_{n+1}) which is consistent with all previous assignments, and consequently the assignment can be extended. Thus, we have the following Theorem [137]:

Theorem 5.2 (van Beek) *Let R be a path-consistent binary constraint network. If there exists an ordering of the domains $D_1 \ldots D_n$ of R such that the relations are row convex, the network is minimal and globally consistent*

For continuous constraints, the convexity property has so far only been applied to STPs, where constraints are *unit difference constraints* of the form $x_i - x_j < c$. This class of constraints is closed under intersection and composition, so that path-consistency algorithms can be implemented without any more complex representations of solution domains. Another problem is that it is sometimes not obvious how to verify that constraints are in fact convex. The representation techniques based on 2^k-trees we gave earlier allow implementation of path-consistency algorithms for continuous constraints of arbitrary form, and we also present later a simple algorithm for reliably checking the convexity of regions represented by 2^k-trees.

5.2.1 Convex binary relations

We define:

Definition 5.3 ***Convex relation***
A relation R_{x_1,x_2} is convex if it determines a convex solution space in the domain $D_{x_1} \times D_{x_2}$

A CCSP is called convex when all its relations are convex.

Definition 5.4 *: **(x)-Intersection***
The (x_i)-intersection of two relations, $R^1_{x_1,x_2}$ and $R^2_{x_1,x_2}$, is the intersection of their projections over the x_i axis ($i \in \{1,2\}$).

The continuous counterpart of van Beek's Lemma (5.1) on row convexity can then be stated as follows:

Lemma 5.2 *If a finite collection, $\mathcal{F}$, of convex relations in $\Re^2$ is such that every pair of relations has a non null (x)-intersection, then the (x)-intersection of all these relations is not null (i.e., there exists at least one value v for x so that each relation $R_{x,y}$ contains a point (v, y_i), where y_i is a possible value for y).*

Proof. The demonstration follows immediately from Helly's Theorem in one dimension (Figure 5.1).
We can now propose the following generalization of van Beek's results to continuous domains:

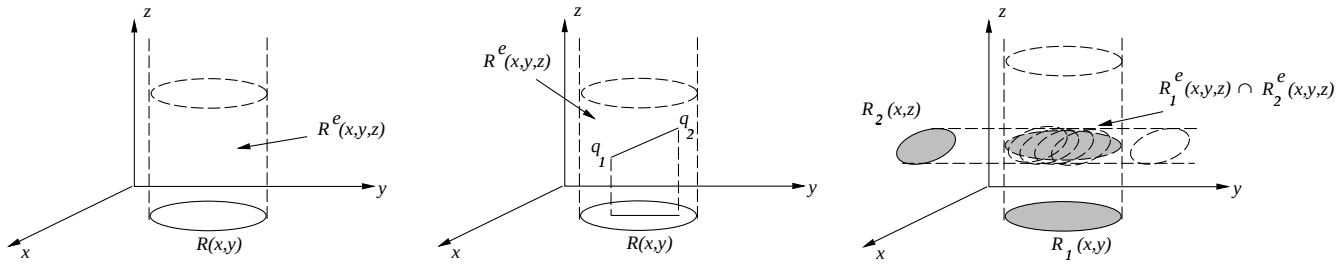


Figure 5.2: *The convexity property is conservative with respect to composition of constraints*

Theorem 5.3 *A binary constraint network which is convex and path-consistent is minimal and decomposable*

Proof. The demonstration is similar to the one proposed in [137] for discrete problems. It is based on the fact that the convexity of the feasibility space guarantees the convexity of any unary projection ((x)-projections). Since unary projections of constraints are convex and since path-consistency guarantees that each pairwise intersection of unary projection is non null, Helly's Theorem in $\Re$ can be applied to prove that a set of k consistently instantiated variables can be extended to a $(k+1)^{st}$ variable $\Box$.

The convexity property is conservative with respect to intersection and projection of constraints. Let us show that this is also the case for composition. For the sake of simplicity, we consider the case of binary convex relations (the extension to n-ary convex relations can be done in a strictly similar way). A variable z taking its values within a convex — possibly unbounded — interval I_z of $\Re$, we define the extension $R^e(x,y,z)$ of a binary relation $R(x,y)$ to the volume $x \times y \times z$ as the exhaustive set of triplets (X,Y,Z) such that X and Y satisfy $R(x,y)$ and $Z \in I_z$ (see Figure 5.2, -A-).

When $R(x,y)$ is convex its 3D extension $R^e(x,y,z)$ is also convex. In effect, any pair of points (q_1,q_2) in $R^e(x,y,z)$ can be joined by a line segment entirely contained in $R^e(x,y,z)$ (otherwise, this would mean that the projection of $\overline{q_1,q_2}$ over the plane $x \times y$ is not entirely contained in $R(x,y)$ which contradicts the fact that $R(x,y)$ is convex) (see Figure 5.2, -B-).

The composition of two binary relations, $R_1(y,x)$ and $R_2(x,z)$, can be defined as the intersection of $R_1^e(y,x,z)$ and $R_2^e(x,z,y)$ (see Figure 5.2, -C-).

When both $R_1(y,x)$ and $R_2(x,z)$ are convex, we know that $R_1^e(y,x,z)$ and $R_2^e(x,z,y)$ are also convex and thus, that $R_1^e(y,x,z) \bigcap R_2^e(x,z,y)$ is convex. Hence, the convexity

property is also conservative with respect to composition.

When a CCSP is convex, it is therefore guaranteed that its path-consistent counterpart remains so and the following *a priori* results can be stated:

Corollary 5.1 *Path-consistency computes the minimal and decomposable network representation of any convex binary CCSP.*

5.2.2 Convex n-ary relations ($n > 2$)

N-ary CCSPs can be translated into ternary ones without loss of information. The generalization of the results on convexity to the case of ternary networks seems straightforward a priori.

In the case of ternary constraints, Lemma 5.2 can be used to prove the decomposability of the constraint network only if each pair of ternary relations has a non null (x)-intersection. Indeed, building a proof analogous to the one used for demonstrating Theorem 5.3 requires proving strong k-consistency of the network for every k. Given that the network is strongly (k-1)-consistent, the following assertion must therefore be satisfied:

$$(\forall (i,j) \in \{1, \ldots k-1\}), R(X_k, X_i, X_j) \tag{5.1}$$

Verifying assertion (5.1) amounts to showing that the interval on x_k, $X_k^1 = \prod_{x_k} R(x_k, X_{i_1}, X_{j_1})$, intersects the interval on x_k, $X_k^2 = \prod_{x_k} R(x_k, X_{i_2}, X_{j_2})$, for any combination of i_1, i_2, j_1, and j_2 in $\{1, \ldots, k-1\}$. Each subset of *five* variables $(x_{i_1}, x_{i_2}, x_{j_1}, x_{j_2}, x_k)$ has therefore to be consistently labeled. This requirement corresponds to the definition of strong 5-consistency. Note that in the particular case where each pair of ternary constraints has two variables in common (i.e., $i_1 = i_2$, j_1 or j_2) the number of variables that must be consistently labeled reduces to four and strong 4-consistency is sufficient for the network to be decomposable. Hence, a possible generalization of Theorem 5.3 could be as follows:

Theorem 5.4 *A ternary constraint network which is convex and strongly 5-consistent is minimal and decomposable.*

Since the convexity property is conservative with respect to relaxation on 2^k-trees, an algorithm enforcing strong 5-consistency would consequently compute the globally consistent network of any convex ternary problem.

Unfortunately, even if theoretically correct, this result has no significant practical impact. Indeed, and analogously to the case of discrete row convex problems, enforcing 5-consistency may create intermediate constraints of arity *four*, which disables the application of Helly's Theorem to *ternary* networks. To overcome this limitation, van Beek and Dechter [139] have introduced the notion of relational path-consistency for discrete problems:

Definition 5.5 (van Beek & Dechter) *Let $\mathcal{R}$ be a network of relations over a set of variables X, and let $\mathcal{R}_S$ and $\mathcal{R}_T$ be two relations in $\mathcal{R}$, where $S, T \subseteq X$. We say that $\mathcal{R}_S$ and $\mathcal{R}_T$ are relationally path-consistent relative to variable x iff any consistent instantiation of the variables in $(S \bigcup T) - \{x\}$, has an extension to x that satisfies $\mathcal{R}_S$ and $\mathcal{R}_T$ simultaneously. A pair of relations $\mathcal{R}_S$ and $\mathcal{R}_T$ is relationally path-consistent iff it is relationally path-consistent relative to each variable in $(S \bigcap T)$. A network is relationally path-consistent iff every pair of relations is relationally path-consistent.*

By definition, relational path-consistency guarantees for each set of relations having a variable x in common, that the pairwise intersections of their unary projections over the x axis are non empty. Helly's Theorem becomes thereby applicable, which results in the following Theorem [139]:

Theorem 5.5 (van Beek & Dechter) *Let R be a network of relations that is relationally path-consistent. If there exists an ordering of the domains $D_1 \ldots D_n$ of R such that the relations are row convex, the network is globally consistent.*

The problem with this approach is that enforcing relational path-consistency may pose complexity problems. In the following, we examine the reasons behind this complexity and discuss the basis for an alternative generalization.

Relational path-consistency is proposed in [139] to ensure pairwise non null intersection of *unary projections*, with the objective of applying Helly's Theorem in *one dimension* (Figure 5.1). In the case of ternary networks, composing *pairs* of ternary relations (with a variable in common) results in a relation of arity five. Relational path-consistency aims

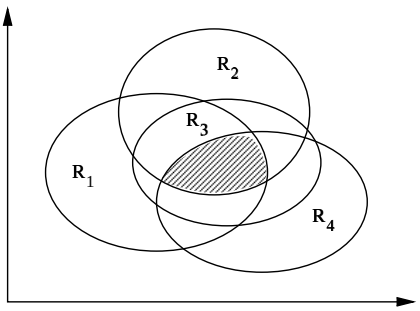


Figure 5.3: *Helly's Theorem in $\Re^2$: if a finite set of binary convex regions is such that each triplet of regions has a non null intersection, then the whole set of regions has a non null common intersection (shaded area)*

at ensuring extensibility properties of an arbitrary subset of variables to a third *single* variable (unary projection). Since arity five is inevitable for ternary constraints, it might be necessary to guarantee that a set of four variables is extensible to a fifth one, so that the relation between the five variables (issued from extended-composition) is not empty. This means that it might be necessary to ensure relational path-consistency for relations of arity four —and recurrently, for relations of arbitrary arity, thus possibly engendering an untractable complexity in the most general case.

The alternative generalization we propose is based on the observation that the extensibility of a ternary set of variables to a *binary* region (rather than to a *unary* one like for relational path-consistency) does not involve relations with arity greater than 3 and thus removes the causes behind combinatorial explosion. This approach implies that Helly's Theorem must be applied in two dimensions rather than one (see Figure 5.3).

For the case of ternary networks, we introduce the notion of (3,2)-relational consistency which guarantees that each *triplet of relations*, with *two variables in common*, has a non null intersection.

Definition 5.6 *Let N be a ternary network of relations over a set of variables X. Let R_{I_1}, R_{I_2} and R_{I_3} be three relations of N. R_{I_1}, R_{I_2} and R_{I_3} are (3,2)-relationally consistent relative to two variables $\{u,v\}$ if and only if any consistent instantiation of the variables in $(I_1 \bigcup I_2 \bigcup I_3) - \{u,v\}$ has an extension to $\{u,v\}$ that satisfies R_{I_1}, R_{I_2} and R_{I_3} simultaneously*

Since (3,2)-relational consistency only requires labels between at most three variables, it does not add to the arity of a ternary constraint network. Provided that each binary

projection is convex, (3,2)-relational consistency enables the application of Helly's Theorem in two dimensions. However, Helly's Theorem only guarantees in these conditions that each pair of variables (binary projections) has a non empty domain. It remains to show that the constraints on each *individual* variable are non empty.

We consider convex networks of relations whose arities are three or less (i.e., ternary relations and hence their binary projections are convex). A relation between three variables i, j, and k is denoted as $R(i,j,k)$. When $i = j$ or $i = k$ or $j = k$, the relations $R(i,j,k)$ is by convention a binary relation.
We present below a brute-force algorithm for enforcing (3,2)-relational consistency. This algorithm takes as input a continuous CSP, $\mathcal{P} = (V, C, D)$. R denotes relations in C.

(3,2)-relational-consistency:
$Q \leftarrow C$
For each ternary tuple $(i,j,k) \in V^3$ and each binary tuple $(u,v) \in V^2$ Do
$R'(i,j,k) \leftarrow R(i,j,k) \oplus \prod_{(i,j,k)} R(i,u,v) \otimes R(j,u,v) \otimes R(k,u,v)$
Until $Q = C$

The operation $R(i,u,v) \otimes R(j,u,v) \otimes R(k,u,v)$ will be called *binary-extension*. Note that each ternary relation takes implicitly into account the binary relations present in the network N (i.e., $R(i,u,v) = R(i,u,v) \otimes R(i,u) \otimes R(i,v) \otimes R(u,v)$).

We now demonstrate the following Theorem:

Theorem 5.6 *For any convex ternary network N, (3,2)-relational-consistency will either decide that the network is inconsistent or else compute an equivalent globally-consistent network of N, in $O(n^5)$ where n is the number of variables of N*

Let us first demonstrate some intermediate results.

Lemma 5.3 *Applied to a ternary network N, the algorithm for (3,2)-relational-consistency computes a ternary network N' such that, if there exists an arbitrary subset of variables in N having no extension to an arbitrary binary region, all relations in N' are empty.*

Proof. Suppose that there exists a subset of k variables, $\{x_1, \ldots x_k\}$, from V having no extension to an arbitray binary region (u,v) (where $u,v \in V^2$). This would

mean that $\prod_{u,v} R(x_1,u,v) \oplus \ldots \oplus \prod_{u,v} R(x_k,u,v) = \emptyset$. According to Helly's theorem in 2D, this situation can occur only in the case where there exists in N at least one triplet of relations $R(x_a,u,v)$, $R(x_b,u,v)$ and $R(x_c,u,v)$ such that $x_a, x_b, x_c \in \{x_1 \ldots x_k\}$ and $\prod_{u,v} R(x_a,u,v) \oplus \prod_{u,v} R(x_b,u,v) \oplus \prod_{u,v} R(x_c,u,v) = \emptyset$. This would mean that there exists in N an empty ternary relation $R(x_a,x_b,x_c)$. The algorithm for (3,2)-relational-consistency computes the closure of N with respect to binary-extension. The fact that there exists in N an empty ternary relation $R(x_a,x_b,x_c)$ means that each relation involving variables x_a, x_b or x_c will become empty in N' (since: $R(x_a,u,v) = R(x_a,u,v) \oplus R(x_a,x_b,x_c) \otimes R(u,x_b,x_c) \otimes R(v,x_b,x_c)$). Moreover, each variable of N is involved in a ternary relation in which x_a, x_b or x_c participate (closure). This means that each ternary relation (in which an arbitrary variable participates) will become empty in N'. Hence, the fact that an arbitrary subset of N's variables admits no extension to an arbitrary binary region (u,v) from V^2 will necessarily result in an empty $(3,2)$-relationally consistent network N' $\Box$.

In order to show that non empty labels on each pair of variables implies global consistency, we introduce a dual network $N_d(V_d, C_d, D_d)$ such that:

- $V_d = \{\alpha_1, \ldots \alpha_m\}$. A variable of V_d, α_j, represents a pair of variables in the original network (an element $(x_{j,1}, x_{j,2})$ of V^2 so that $x_{(j,1)} \neq x_{(j,2)}$)
- a domain of D_d is an element of D^2
- a relation between two variables, α_u and α_v, of V_d is the relation $R(x_{(u,1)}, x_{(u,2)}, x_{(v,1)}, x_{(v,2)})$ issued from the composition of the relations between $x_{(u,1)}, x_{(u,2)}, x_{(v,1)}, x_{(v,2)}$ in the original problem.

Note that N_d is now a *binary* constraint network.

Lemma 5.4 *Let N be a (3,2)-relationally consistent ternary network and let N_d be its dual representation. The following propositions are verified:*

i. each solution of N corresponds to a solution of N_d

ii. each solution of N_d corresponds to a solution of N

Proof. Case *i*. The demonstration is straightforward: if $S = \{X_1, \ldots X_n\}$ is a solution of N (i.e., is a consistent instantiation for the variables $\{x_1, \ldots x_n\}$), then, by definition of the dual network N_d, the set of binary regions $\{(X_i \otimes X_j) \| i, j = [1 \ldots n], i < j\}$ is a solution for the variables $\alpha_k = (x_i, x_j)$ of N_d. □

Case *ii*. A value for a node α_j of N_d is a binary region I_j involving the variables $x_{(j,1)}$ and $x_{(j,2)}$. We denote as $\prod_{x_{(j,1)}} I_j$, the projection of I_j over the axis $x_{(j,1)}$. Let us first show that $\prod_x I_u = \prod_x I_v \forall x \in V$ and $\forall u, v \in V_d$ (i.e., each instantiation of N_d corresponds to a unique valuation for the variables of N). Suppose that there exist two nodes of N_d, $\alpha_a = (x_i, x_u)$ and $\alpha_b = (x_i, x_v)$ so that $\prod_{x_i} I_a \neq \prod_{x_i} I_b$, this would mean that there exists a value l for x_i so that $\prod_{x_i, x_v}^{j:x_1 \ldots x_n} R(l, x_v, j)$ is not empty while $\prod_{x_i, x_v}^{j:x_1 \ldots x_n} R(l, x_u, j)$ is. This implies that there exist at least two variables y and z so that $R(l, x_v, y) \otimes R(l, x_v, z) = \emptyset$ and $R(l, x_u, y) \otimes R(l, x_u, z) \neq \emptyset$. This situation is incompatible with (3,2)-relational consistency since it means that there exist three values, l for x_i, l' for y and l'' for z so that the relation $R(x_i, y, z)$ has no binary extension to (x_u, x_v). Consequently, each solution of N_d corresponds to a unique valuation, $S = X_1, X_2, \ldots, X_n$, of N's variables. Let us now show that S is also a solution for N. If S is not, there exists in S a subset of values $\{X_1, X_2, ..X_k\}$, inconsistent with respect to the relations of N involving variables $\{x_1, \ldots x_k\}$. This means that the valuation $\{(X_1, X_k), (X_2, X_k) \ldots (X_{k-1}, X_k)\}$ is inconsistent for the variables $\{(x_1, x_k), (x_2, x_k) \ldots (x_{k-1}, x_k)\}$ of N_d, which contradicts the hypotheses □.

Lemma 5.5 *Let N be a convex ternary network. Let N' the network computed from N by (3,2)-relational consistency and let N_d be its dual representation. If N contains a subset of variables having no extension to an arbitrary binary region (u, v) $(u, v \in V)$, then all relations in N_d are empty, else every assignment of variables in N_d can be consistently extended and N_d is globally consistent.*

Proof. As shown in the proof of Lemma 5.3, if an arbitrary subset of N's variables has no binary extension to an arbitrary binary region (u, v) $(u, v \in V)$, the algorithm (3,2)-relational-consistency will detect an empty ternary relation in N and propagate it until all the relations in N' become empty. When translating N' into its dual form, all relations in C_d will also be empty. Now suppose that each subset of N''s variables has a non empty extension to an arbitrary binary region (u, v) $(u, v \in V)$. According to

Lemma 5.3, we know that no relation in N' is empty. This also means that no relation in N_d is empty: suppose that there exists an empty relation in N_d, between two nodes (x_i, x_j) and (x_u, x_v), this means that the revision step: $R(x_i, x_j, x_j) = R(x_i, x_j, x_j) \oplus \prod_{x_i,x_j} R(x_i, x_u, x_v) \otimes R(x_j, x_u, x_v) \otimes R(x_j, x_u, x_v)$ would have produced an empty relation in N', which contradicts the hypothesis.

Let us now show that, in the case where the relations of N' are not empty, each consistent instantiation of three variables in N_d can be consistently extended to a fourth one.

Suppose that this is not verified. This would mean that there exists a set of three relations of N_d and a variable α_4 of N_d such that:

$$\prod_{(\alpha_4)} R(\alpha_1, \alpha_4) \otimes R(\alpha_2, \alpha_4) \otimes R(\alpha_3, \alpha_4) = \emptyset \tag{5.2}$$

Assertion (5.2) amounts to saying that there exists in N_d, a set of three variables $\{\alpha_1 = (x_1, x_2), \alpha_2 = (x_3, x_4), \alpha_3 = (x_5, x_6)\}$ having no extension to a fourth variable $\alpha_4 = (x_7, x_8)$, which is equivalent to saying that there exists a set of (at most) six values in N, $\{X_1, X_2, \ldots X_6\}$, so that the variables $\{x_1, x_2, \ldots x_6\}$having no binary extension to $\{x_7, x_8\}$:

$$(\prod_{(x_7,x_8)}^{i:1..6} R(x_7, x_8, X_i) = \emptyset). \tag{5.3}$$

Since N' is (3,2)-relationally consistent and since no relation in N' is empty, each triplet of relations $R(x_7, x_8, x_a), R(x_7, x_8, x_b)$ and $R(x_7, x_8, x_c)$ where $(a, b, c) \in [1..6]$ has a non-null intersection. According to Helly's Theorem in $\Re^2$, $\prod_{(x_7,x_8)}^{i:1..6} R(x_7, x_8, X_i)$ is consequently not empty, which contradicts assertion (5.3).

Suppose now that no relation of N_d is empty and that $k-1$ variables of N_d have been consistently instantiated. We must demonstrate that this consistent sub-instantiation can be extended consistently to a k^{th} variable. N' is (3,2)-relationally consistent. We know that each consistent instantiation of three variables in N_d can be consistently extended to a fourth one. This guarantees that each subset of three N_d's relations, $R(\alpha_k, \alpha_a), R(\alpha_k, \alpha_b), R(\alpha_k, \alpha_c)$, (where $(a, b, c) \in [1..k-1]$) has a non null projection over α_k. Since each relation $\prod_{\alpha_k} R(\alpha_k, \alpha_i)$ is a convex, non-empty region of $\Re^2$ (variables $\alpha_i, i \in [1..k-1]$ are instantiated), Helly's Theorem is applicable and guarantees that $\prod_{\alpha_k}^{i:[1..k-1]} R(\alpha_k, \alpha_i) \neq \emptyset$. This means that α_k can be instantiated consistently. This result holds for an arbitrary k, hence N_d is globally consistent $\Box$.

Theorem 5.6 follows immediately from Lemmas 5.4 and 5.5. As to complexity considerations: the number of relations checked for binary-extensibility is initially in $O(n^3 + n^2) = O(n^3)$. Each time a relation is modified, $O(n^2 + n) = O(n^2)$ new binary-extensions are computed. The global complexity of (3,2)-relational-consistency is therefore $O(n^5)$ $\Box$.

Given that k variables $\{x_1, x_2 \ldots x_k\}$ of N have already been instantiated, finding a value for a third variable x_{k+1} is always possible: it amounts to finding a value, in N_d, for a node $(x_{k+1}, x_j), j = [1..k]$. A possible backtrack-free instantiation procedure for deriving the solutions of N would be as follows:

1. Choose a value X_1 of x_1 that satisfies $R(x_1)$
2. For $i \leftarrow 2$ to n do
3. $\quad I_i \leftarrow \bigcap_{j:1..i-1} \prod_{x_i} R(xi, X_1, X_j)$
4. $\quad X_i \leftarrow$ choose a value for x_i in I_i

Comparison with the discrete case — 2^k-trees and backtrack-free search:

It is worth mentioning that the results on n-ary constraints (see section 5.2.2) are not directly transferable to discrete domains. In effect, ensuring backtrack-free search in 3D requires convexity conditions to be verified in $\Re^2$ at least (rather that $\Re$). Given a ternary discrete CSP represented by (0,1) matrices we can demonstrate that, (3,2)-relational consistency guarantees global consistency only if the intersection of the binary projection of two relations yields a convex non-empty feasible domain. Row convexity is not sufficient to guarantee this property, since it only provides a convexity guarantee in one dimension:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \oplus \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \longrightarrow \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Translated to the continuous case, the example above corresponds to the situation where two convex regions (dashed ellipsis) yield an empty intersection — this intersection falls below the resolution of the matrix.

Using the discretized 2^k-tree representation of continuous constraints situations can occur where, intersecting the binary projections of several ternary convex relations results

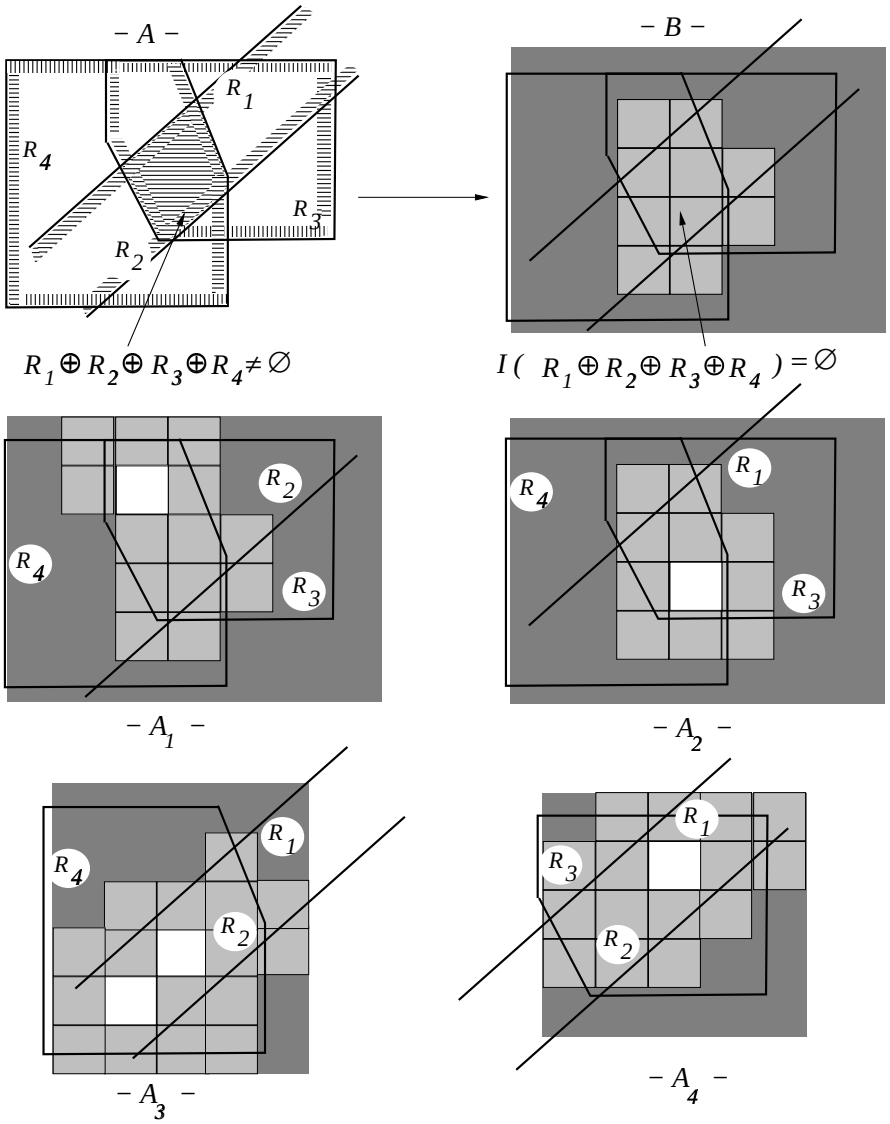


Figure 5.4: *When the precision of the decomposition is insufficient, intersecting the binary projections of a set of ternary convex relations can result in an empty $\mathcal{I}(S)$ representation*

in an empty domain. Figure 5.4 gives an example where the 2^k-tree approximations of four binary relations (Figure 5.4, -A-) are such that:

- the intersection of each triplet of relations has a non null inner content approximation (i.e. each intersection of three relations includes at least one white node, see Figure 5.4, -A_1, A_2, A_3 and A_4-)
- the intersection of the four relations has an empty inner content approximation (no white node, see Figure 5.4, -B-),

However, it is important to notice that contrary to the discrete case, the empty intersection is not due to the topology of the domain but to an inappropriate level of resolution. Therefore, it suffices to refine the decomposition to identify a non empty intersection.

To summarize, the example of Figure 5.4 shows that using the $\mathcal{I}(S)$ representation (inner content) of continuous constraints, a given step of the instantiation process may produce an empty domain (no white node) even if the network is (3,2)-relationally consistent and all the original relations are convex. However, since the $\mathcal{O}(S)$ representation of a convex relation is proved to be connected and non-empty (see proposition 3.3), discovering an empty intersection during the instantiation process does not invalidate the guarantee of backtrack-free search. It only means that further refinements of the decomposition are required for identifying the solution. Refinements which, due to the nature of the domain, cannot be performed on discrete relations.

On the other hand, extending the row-convexity property to 2D, so that Helly's theorem can ensure the binary extensibility condition, would dictate that each discrete ternary relation yields a universal matrix (with 1s only) as binary projection:

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \oplus \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

which is probably too restrictive for practical use.

5.3 Partially convex continuous relations

Constraint convexity is a rather strong condition, but it turns out that weaker forms of convexity are often sufficient to satisfy the conditions of a globally consistent labeling. We

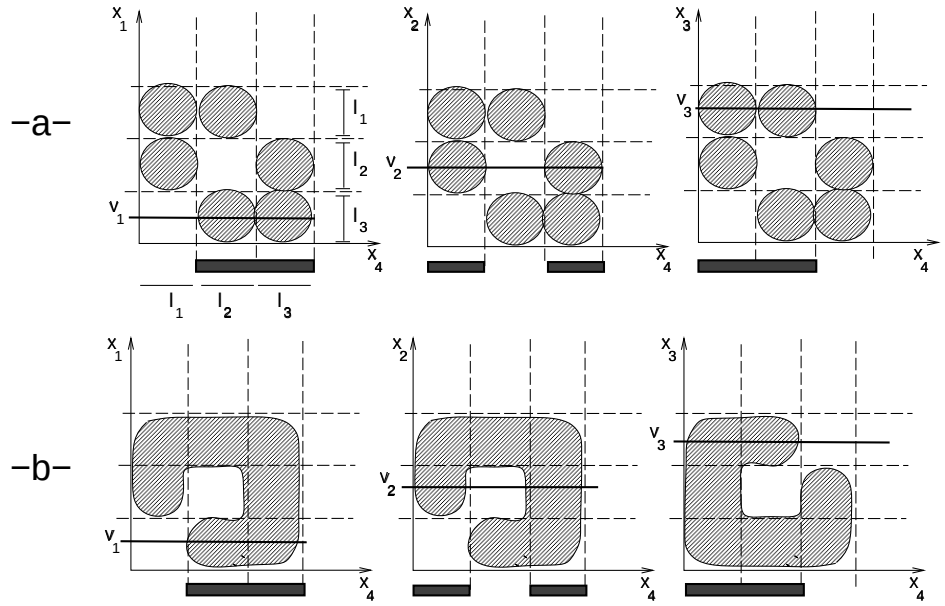


Figure 5.5: *Path-consistency is not always sufficient for determining inconsistencies in simply connected solution spaces*

recall the definitions of some basic topological properties needed to characterize continuous solution spaces. These definitions are taken from [67] and [110].

Definition 5.7 (VNR encyclopedia) *A point set D is path-connected if any two points p and q in D can be joined by a path in D.*

Definition 5.8 (VNR encyclopedia) *A point set D is simply-connected if any closed curve in D can be contracted inside D to a point.*

In 3D, this property requires that any closed surface must be contractable to a point.

Definition 5.9 (Preparata-Shamos) *A domain D in E^d is convex if, for any two points p and q in D, the line segment $\overline{pq}$ is entirely contained in D. (E denotes the Euclidean space).*

There exists a continuous map of an interval into a path connected region. This characterizes the weaker form of convexity that can be exploited. A disconnected (non path connected) solution space generally yields non convex projections over the axis involved: path connectivity is therefore a *necessary* requirement for extending to continuous spaces the results presented above. However, when we are looking for global consistency, it is not

sufficient. Convex outer projections aid in determining a tighter consistent approximation of the solution space but do not guarantee that the resulting bounds are minimal. This is due to the fact that even if the outer projection is convex, this property is not necessarily preserved for a subprojection of the solution space. The simple example of (Figure 5.5, -a-) is mapped from a standard unsolvable graph coloring problem with three colors and four variables. Each variable can take its values within one of three intervals I_1, I_2 and I_3 and each variable has to be assigned values from different intervals. The problem of Figure 5.5 is path-consistent but not 4-consistent —it admits no solution: the horizontal lines in Figure 5.5 show a sequence of assignments $(x_1 = v_1, x_2 = v_2, x_3 = v_3)$ leading to an inconsistency. In this example, the constraints do not determine path-connected regions but it is possible to construct a variant of the same problem with simply connected regions by just tearing and pasting different disconnected regions (see Figure 5.5, -b-). Consequently, neither path connectivity nor simple connectivity characterize convexity properties well enough to extend van Beek's results on minimality and decomposability.

For this reason, we introduce a new category of partial convexity called (x)-*convexity*. This property is more restrictive than path and simple connectivity but guarantees the convexity of any subprojection.

Analogously to the case of row convex CSPs, we will also show that weaker forms of (x)-convexity (namely directional (x)-convexity) provide backtrack free resolution schemes for a large variety of problems with simply connected solution spaces. It is worth mentioning that the results we present in the following are all *a posteriori* results since the partial convexity we introduce may disappear during propagation.

We start by discussing the case of binary constraints.

5.3.1 Partially convex binary relations

We define:

Definition 5.10 : ***(x)-Convexity***

Let R be a binary relation defined by a set of algebraic or transcendental constraints on two variables x_1, x_2. R is said to be x_k-convex in the domain D_{x_k} if it defines a connected region r and if any two points q_1 and q_2 of r such that the segment $\overline{q_1 q_2}$ is parallel to x_k, $\overline{q_1 q_2}$ is entirely contained in r.

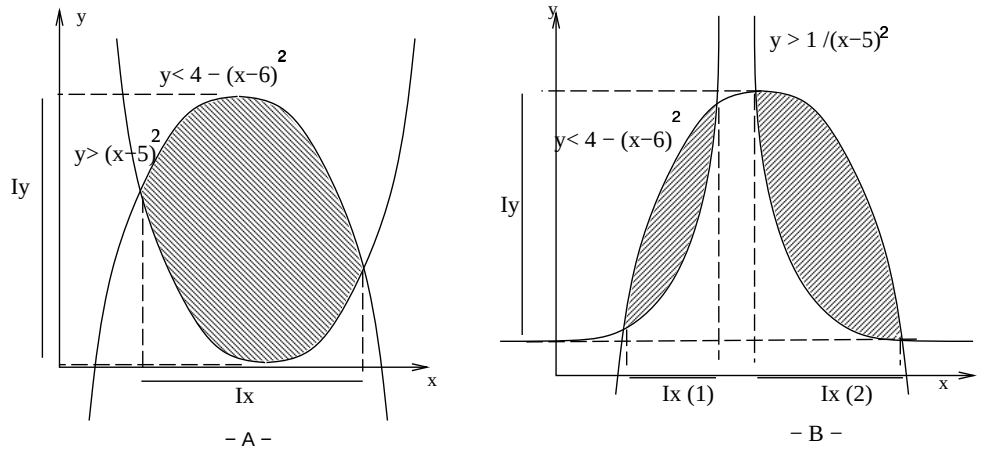


Figure 5.6: *Figure -A- illustrates a binary relation R_{xy} given intentionally by the two inequalities $y > (x-5)^2$ and $y < 4-(x-6)^2$. R_{xy} is convex (and consequently y- and x-convex). The projection of R_{xy} respectively over the x and y axes yields single bounded intervals (respectively in I_x and I_y) for each value of x and y. In Figure -B-, the relation R_{xy} is given intentionally by the constraints $y > 1/(x-5)^2$ and $y < 4-(x-6)^2$. Relation -B- is only y-convex since it is not path-connected (its projection over the x-axis yields two distinct intervals I_{x1} and I_{x2}).*

A network is said to be (x)-convex if each of its relations R_{xy} is (x)-convex.

Convexity imposes a more restrictive condition than (x)-convexity: the former property allows q_1 and q_2 to be chosen in the domain with no restriction. The latter property involves choosing q_1 and q_2 within the set of points belonging to lines parallel to the x axis. A convex region is necessarily (x)-convex but the converse is not true.

Figure 5.6 illustrates the notion of convex and x-convex relations.

Lemma 5.6 *Let R be a relation defined by a set of algebraic or transcendental constraints on two variables x_1, x_2. R is x_k-convex if and only if any subprojection of R over the x_k axis yields a single convex interval.*

Proof: The proof by contradiction is straightforward. In the following D_i denotes the domain of variable i. Suppose first that R is x_k-convex and that there exists a subdomain, $Sub = S_1 \times ..S_{k-1} \times S_{k+1}.. \times S_n$, of $D_1 \times ..D_{k-1} \times D_{k+1} \times ..D_n$ such that $\prod_{Sub}^{x_i}$ yields two distinct intervals $[l_1..u_1]$ and $[l_2..u_2]$ (with $[l_1..u_1] \bigcap [l_2..u_2] = \emptyset$). Let $a' = (a_1.., a_k, ..a_n)$ be a point of $S_1 \times ..S_{k-1} \times [l_1..u_1] \times S_{k+1}..S_n$ and $a' = (a_1.., a'_k, ..a_n)$ be a point of $S_1 \times ..S_{k-1} \times [l_2..u_2] \times S_{k+1}..S_n$. The segment $\overline{aa'}$ is parallel to x_k by construction. $\overline{aa'}$ is not entirely contained in R since the segment $\overline{a_k a'_k}$ is not entirely contained in the projection $\prod_{Sub}^{x_i}$ of

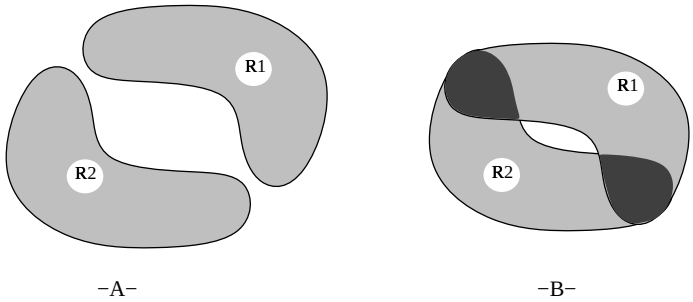


Figure 5.7: *The intersection of two (x)-convex relations is not necessarily (x)-convex: intersecting R_1, R_2, yields two disjoint regions (dark regions, -B-) and hence, $R_1 \oplus R_2$ is not (x)-convex*

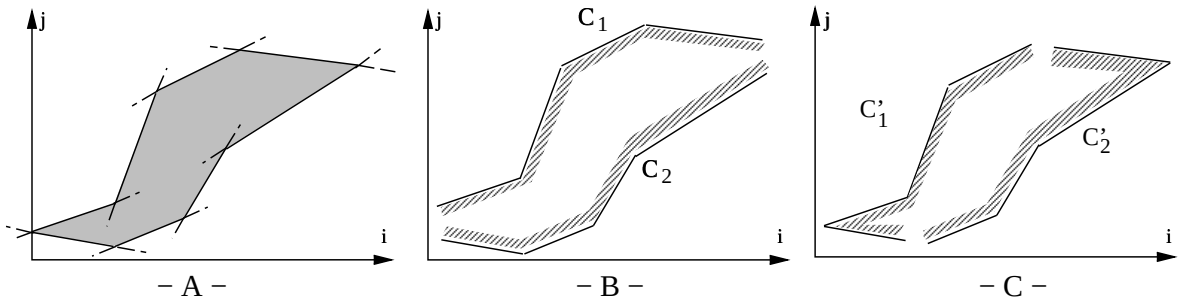


Figure 5.8: *Figure -A- shows the polygonal solution space of a linear problem which is both (i)- and (j)-convex but not convex: it is monotone wrt i (respectively j) since its boundaries can be decomposed into two monotone chains C_1 and C_2 (respectively C'_1, C'_2) monotone wrt i (respectively j)*

R. Hence there exists a pair of points a and a' such that $a \in r$, $a' \in r$ and the segment $\overline{aa'}$ is not entirely contained in R. This contradicts the definition of x_k-convexity. Conversely, suppose that R is not x_k-convex but that any subprojection of R over x_k yields a convex interval. Since R is not x_k-convex, there exists a pair of points $a = a_1 \ldots, a_k, \ldots a_n$ and $a' = a_1 \ldots, a'_k, \ldots a_n$ of R such that the segment $\overline{aa'}$ is not entirely contained in R. The straight line l, parallel to x_k and containing the points a and a' must therefore contain at least one point $a'' = a_1 \ldots, a''_k, \ldots a_n$ so that $a'' \notin r$. Hence, projecting R over x_k in the subdomain represented by l will yield two non overlapping intervals $[a_k a''_k[$ and $]a''_k a'k]$. This contradicts the fact that any subprojection yields a single convex interval □.

Lemma 5.2 generalizes straightforwardly to (x)-convex binary regions. This allows the formulation of the following continuous counterparts of Theorem 5.2 proposed by van Beek in [137]:

Theorem 5.7 *A binary constraint network which is (x)-convex and path-consistent is minimal and decomposable*

As illustrated in Figure 5.7, the (x)-convexity property is non conservative with respect to intersection. Hence, the global consistency property stated by Theorem 5.7 is only guaranteed for the *a posteriori* network computed by path-consistency.

Monotone and functional relations

The (x)-convexity property shares important analogies with the notion of monotonicity as defined in Euclidean geometry. A chain is defined as a set of vertices connected with segments of lines. It is said to be monotone with respect to a straight line l if a line orthogonal to l intersects C in exactly one point. A simple polygon is said to be monotone if its boundaries can be decomposed into two chains monotone with respect to the same line.

The definitions given for polygonal objects can be extended to general curves as follows: a curve C of $\Re^2$ will be called monotone with respect to a straight line l, if a line orthogonal to l intersects C in exactly one point, and a region r of $\Re^2$ will be said monotone if its boundaries can be decomposed into two curves monotone with respect to the same line. This generalized definition of monotonicity is equivalent to (x)-convexity. A linear problem having a polygon monotone[2] wrt. x as representation in a $(x \times y)$ Cartesian coordinate system, is consequently (y)-convex. Note that even if verified for both axes, the (x)-convexity property remains a requirement weaker than convexity (see Figure 5.8).

As another example, the solution spaces defined by single functions also verify (x)-convexity properties: the curve defined by a function f(x,y), functional in x, is by definition monotone wrt x.

Directional (x)-convexity

In the case where a discrete network does not satisfy the row-convexity property, van Beek shows that directional row-convexity remains a useful property for exhibiting backtrack-free solutions. Similar results generalize to the case of (x)-convex relations. The following

[2]Testing for the monotonicity of a polygon can be done in $O(N)$ where N is the number of edges.

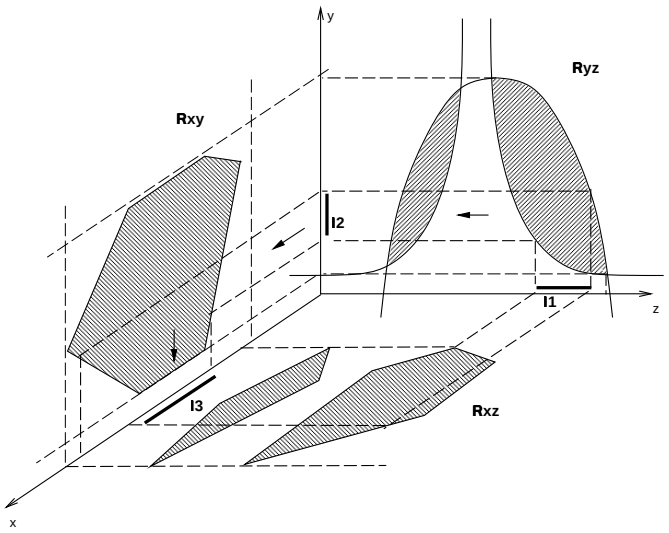


Figure 5.9: *This CCSP with three variables x,y,z and three relations R_{xy}, R_{xz} and R_{xy} is path-consistent but not convex since R_{yz} is not z-convex. A backtrack-free instantiation can however be found by first assigning I_1 to z, then I_2 to y and finally I_3 to x. The interval I_3 is the x-intersection of R_{xz} and R_{xy}.*

Theorem states that a partial (x)-convexity of the network is sufficient to ensure that a solution can be determined without backtracking.

Theorem 5.8 *Let N be a path-consistent binary constraint network. If there exists an ordering of the variables $x_1, \ldots x_n$ such that each relation of N R_{x_i,x_j}, $1 \leq j \leq i$, is x_i-convex, then a consistent instantiation can be found without backtracking.*

Proof: We denote as $\prod^i_{X_j} R_{ij}$ the projection over the i axis, in the domain X_j, of the relation R_{ij}. The backtrack-free instantiation algorithm proposed by van Beek (algorithm *instantiate* in [137]) generalizes as follows from the constructive proof of Theorem 5.7:

1. Choose a value X_1 of x_1 that satisfies R_{11}
2. For $i \leftarrow 2$ to n do
3. $\quad X_i \leftarrow$ choose a value in $\bigcap_{j=1\ldots i-1}(\prod^i_{X_j} R_{ij})$

where $\bigcap$ denotes interval intersection. Figure 5.9, illustrates the notion of directional (x)-convexity. Step 3 of the algorithm exploits the i-convexity property along with path-consistency to ensure that interval intersection will return a single non-empty interval for x_i (X_i is then chosen within this interval). We can note that this step requires the

Find-Order($\mathcal{T}$, n):

1 $L \leftarrow \{1 \ldots n\}$;
2 **for** $m \leftarrow n$ to 1 **do**
3 *find a* $j \in L$ *for which* $\forall i \in L, T(i,j)$ *is (x)-convex*;
4 **if** *no such* j *exists* **then** *return Fail*
5 **else** *put variable* x_j *at position* m *in the ordering*
6 $L \leftarrow L - \{j\}$ **fi** **od**

Figure 5.10: *Algorithm for finding a directionally (x)-convex ordering of variables*

i-convexity only for the relations R_{ij} where j denotes the variables already instantiated $(0 < j < i)$.

Ordering variables

Directional row convexity imposes ordering conditions on both variables and variable domains. Since discrete CSPs do not have the strictly ordered domains characterizing continuous CSPs, the fact that a constraint network is row convex can sometimes be hidden. This is obviously not the case for continuous CSPs concerning (x)-convexity. In order to apply the previously described results on directional (x)-convexity, it remains to show how an (x)-convex ordering of the variables can be found.

In [139], van Beek and Dechter propose an algorithm for determining a directional row-convex ordering for discrete problems. This algorithm can be directly mapped into the one in Figure 5.10 for determining an (x)-convex ordering.

The algorithm `Find-Order` takes as input a list of quadtrees $\mathcal{T}$ and the number n of variables involved in the problem. The only difference with the algorithm proposed in [139] relates to the fact that finding an appropriate ordering for the values in the domains is not necessary in the case of continuous relations. Termination and correctness can be proved in a strictly similar way to [139]. In the worst case, step 3 requires n quadtrees to be checked for (x)-convexity. We show in the next section that the complexity of testing the (x)-convexity of a quadtree is in $O(N.log_4(N))$ where N is the number of feasible nodes in the quadtree. The complexity of `Find-Order` is consequently in $O(n.Nlog_4(N))$ where n is the number of variables and N is the number of feasible nodes in the quadtree of maximal size.

5.3.2 Partially convex n-ary relations

We now examine how, analogously to the case of binary constraints, a less restrictive convexity condition can be defined for n-ary constraints. We first propose the following generalization of the (x)-convexity property:

Definition 5.11 *: $(x_1, \ldots x_k)$-Convexity*
Let R be an n-ary relation between n variables $x_1 \ldots x_n$. R is said to be $(x_1, \ldots x_k)$-convex in the domains $D_1 \times \ldots D_{x_k}$ if and only if it defines a connected region r and if for any two points q_1 and q_2 of r, such that the segment $\overline{q_1 q_2}$ is on a plane parallel to $x_1 \ldots \times x_k$, $\overline{q_1 q_2}$ is entirely contained in r

Informally, this means that a relation is $(x_1, \ldots x_k)$-convex if any subprojection over the subset $(x_1, \ldots x_k)$ yields a convex k-ary region. In the case of networks of arity r, the composition of two maximal arity constraints having at least one variable in common, results in a relation of arity $2r - 1$. In analogy to the case of ternary networks, we observe that the extension of an r-ary set of variables to a region of arity $r - 1$ does not involve relations with arity greater than r. To apply Helly's Theorem we must introduce the notion of (r,r-1)-relational consistency which guarantees that each set of r relations having $r - 1$ variables in common has a non null intersection:

Definition 5.12 *Let N be a network of relations over a set of variables X, of arity r. Let $R_{I_1}, \ldots R_{I_r}$ be r relations of N. The relations $R_{I_j}, j : [1 \ldots r]$ are (r,r-1)-relationally consistent relative to $r - 1$ variables $\{u_1, \ldots u_{r-1}\}$ iff any consistent instantiation of the variables in $(\bigcup_{j:[1\ldots r]} I_j) - \{u_1 \ldots u_{r-1}\}$ has an extension to $\{u_1, \ldots u_{r-1}\}$ that satisfies $R_{I_j}, j : [1 \ldots r]$ simultaneously*

Hence, an alternative generalization to Theorem 5.7 could be as follow:

Theorem 5.9 *Let N be a constraint network of arity r at most, $(x_1, \ldots, x_{r-1})$-convex. If N is (r,r-1)-relationally consistent, then it is globally consistent.*

Proof. The proof is similar to the one given for Theorem 5.6. In the general case, the nodes in the binary dual representation of N represent $(r-1)$-ary subsets of N's variables.

Directional (x,y)-convexity

Theorem 5.8 shows that the solution of certain partially (x)-convex binary CCSPs can be derived backtrack-free. We now show that similar results can be stated for ternary CCSPs.

Suppose that a CCSP is (3,2)-relationally consistent and that k of its variables, $x_1, x_2, \ldots x_k$, have been consistently instantiated. In order to derive backtrack-free a consistent value for a $k+1^{st}$ variable, it suffices that each binary relation between x_{k+1} and x_i (with $i \leq k$) is (x_{k+1})-convex and non empty. In effect, (3,2)-relational consistency guarantees the extensibility of each triplet of variables to a fourth and fifth one. This means that for each triplet of values X_a, X_b, X_c there exists a value for x_{k+1} and hence each pair of values X_a, X_b (or X_b, X_c or X_a, X_c) is extensible to x_{k+1}. Therefore, according to Helly's Theorem in one dimension, if each relation $R(x_{k+1}, x_i)$ is (x_{k+1})-convex and non empty, it is guaranteed that $\bigcap_{i:1..k} \prod_{x_{k+1}} R(x_{k+1}, x_i)$ is not empty.

The condition for each binary relation $R(x_{k+1}, x_i)$ to be non-empty is either that there exists a unique non-empty ternary relation involving x_{k+1} and x_i, or that each ternary relation involving the variables x_{k+1} and x_i is non empty and (x_{k+1}, x_i)-convex. Since the (x_{k+1}, x_i)-convexity of a ternary relation involving the variables (x_{k+1}, x_i) implies the (x_{k+1})-convexity of its projection over the plane $(x_{k+1} \times x_i)$, we have:

Theorem 5.10 *Let $\mathcal{N}$ be a (3,2)-relationally consistent ternary constraint network. If there exists an ordering of the variables $x_1, \ldots x_n$ such that for each $i, j : 1 \leq j \leq i$:*

1. *there exists a unique ternary relation $R(x_i, x_j, x_k)$ involving x_i and x_j and $\prod_{(x_i, x_j)} R(x_i, x_j, x_k)$ is (x_i)-convex*

2. *there exist several ternary relations $R(x_i, x_j, x_k)$ involving x_i and x_j and each $R(x_i, x_j, x_k)$ is (x_i, x_j)-convex.*

then a consistent instantiation can be found without backtracking.

The appropriate variable ordering can be derived using a variant of the algorithm *Find-Order* of section 5.3.1, where the test for (x)-convexity is replaced by procedures verifying conditions (1) and (2) of Theorem 5.10.

5.4 Checking for convexity properties

Theoretically, checking for the (x)-convexity or convexity of a solution space involves the identification of convexity deficiencies along its boundary. A procedure working on the actual relations would therefore try to determine their local and absolute extrema. As mentioned in section 5.3.1, there exist efficient (linear) procedures for deciding the monotonicity of simply polygonal solution spaces (linear programming problems). This particular case is based on the enumerability of the edges, which offers a simple means of following a region's contour. Unfortunately, similar solutions do not generalize to arbitrary continuous spaces. Since we want to avoid complex computations for solving sets of simultaneous equalities and inequalities, we will limit ourselves to analyzing the approximate boundaries of a solution space without trying to compute actual extrema.

5.4.1 2^k-trees and convexity

The 2^k-tree decomposition of continuous solution spaces generates stepwise approximations of the boundaries and thus cannot be convex in the strict geometrical definition. However, we have shown in chapter 3 that:

- A convex solution region falling above the limit of resolution has a connected $\mathcal{I}(S)$ representation, with minimal convexity deficencies.
- A convex solution region can yield a disconnected or empty $\mathcal{I}(S)$ representation only in the case where it falls within the limit of resolution. Moreover, when the $\mathcal{I}(S)$ representation is disconnected, a single additional level of decomposition is shown to be sufficient to reconnect the representation.

Let us discuss the practical influence of these convexity deficiencies on the methods and results described earlier. We start by examining the case of binary constraints.

Mathematically, convexity is an interesting property because it is conservative with respect to constraint intersection. This guarantees, *a priori*, that the constraints obtained after constraint propagation are still convex (and thus (x)-convex).

If connected, the quadtree representation of a convex region $R(x, y)$ is necessarily both (y) and (x)-convex. Moreover, we can show that intersecting to (x)-convex regions yields either a (x)-convex region or a set of disconnected (x)-convex regions:

Proposition 5.1 *The intersection of two (x)-convex regions r_1 and r_2 generates either a single (x)-convex region or a set of disconnected (x)-convex regions.*

Proof. Since the intersection of two (x)-convex regions, r_1 and r_2, can create disconnected regions (see figure 5.7), the connectivity of $r_1 \oplus r_2$ is not necessarily preserved. Suppose now that there exist two points q_1 and q_2 of $r_1 \oplus r_2$ such that the segment $\overline{q_1 q_2}$ is parallel to the x axis and is not entirely contained in $r_1 \oplus r_2$. This would mean that there exists a point $q_3 \in \overline{q_1 q_2}$ such that $q_3 \in r_1$ and $q_3 \notin r_2$ (or vice versa). Since q_1 and q_2 are both in r_2, this means that the segment $\overline{q_1 q_2}$ cuts r_2 in more than two points, which contradicts the fact that r_2 is (x)-convex. Hence, an (x)-convexity deficiency of this type cannot be created by intersection. Since this result does not depend upon the connectivity of $r_1 \oplus r_2$, the proposition is proved. □.

Hence, the only risk that presents itself when running the propagation algorithm using the quadtree representation of convex constraints is the risk of disconnection.

In the following, we start by showing that:

- the risk of disconnection due to intersection only occurs when the solution of the CCSP falls within the limit of resolution chosen for the quadtree representation

- a particular class of minimal convexity deficiencies can be identified which precludes the risk of disconnection while intersecting two constraints. The quadtree having minimal deficiencies of this type will be called a *convex quadtree*

Intersecting the quadtree representations of convex relations— risk of disconnection

Suppose that two quadtrees $\mathcal{T}_1$ and $\mathcal{T}_2$, representing two relations R_1 and R_2, only have minimal convexity deficiencies. Since the intersection of two convex regions yields a convex region, any disconnection in $\mathcal{T}_1 \oplus \mathcal{T}_2$ is necessarily due to a convexity deficiency. By definition, each minimal convexity deficiency can be entirely contained in a slice of width ε where ε is the maximal resolution. Hence, a disconnection in $\mathcal{T}_1$ and $\mathcal{T}_2$ can only occur when identifying the solution region of $R_1 \oplus R_2$ requires reaching the limit of resolution.

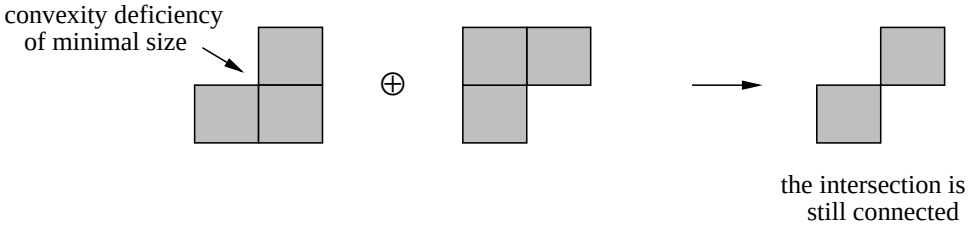


Figure 5.11: *Convexity deficiencies of minimal size do not disconnect the solution space*

For most engineering problems, the maximal useful resolution can be fixed beforehand and solution regions falling oustside these limits can be neglected.

Convex quadtrees

A particular type of minimal convexity deficiency removes completely the risk of disconnection. In effect if we consider quadtrees where each convexity deficiency is delimited by exactly two boundary nodes, no disconnection can be created by intersection (see Figure 5.11).

The side of a node is said to be free if its neighbor along this side is black or does not exist.

We define:

Definition 5.13 *A node of a quadtree $\mathcal{T}(x,y)$ has a y-free (respectively x-free) side if one of its sides, parallel to the y (respectively x) axis, is free*

Definition 5.14 *A quadtree $\mathcal{T}(x,y)$ is convex if it is both x- and (y)-convex, and if any of its x- (respectively y-)free node:*

- *is either adjacent to another x- (respectively y-)free node,*
- *or shares a corner with another x- (respectively y-)free node,*

Note that both x- and (y)-convexity are required: by possibly generating a disconnection, a (y)-convexity deficiency may alter simple-connectivity and hence (x)-convexity. When both x- and (y)-convexity are guaranteed, the maximal convexity deficiencies admissible for a convex quadtree are those corresponding to the discretization steps (one node of minimal size).

Proposition 5.2 *Convexity is closed under quadtree intersection*

Proof. Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two convex quadtrees. Suppose that there exists a y-free node n in $\mathcal{T}_1 \oplus \mathcal{T}_2$ such that n is neither adjacent to, nor shares a corner with another x- (respectively y-)free node. n is necessarily a y-free node of $\mathcal{T}_1$ or $\mathcal{T}_2$ (say $\mathcal{T}_1$). The neighbors of n in $\mathcal{T}_1 \oplus \mathcal{T}_2$ are also the neighbors of n in $\mathcal{T}_1$ (intersection). Since these neighbors do not have y-free sides in $\mathcal{T}_1 \oplus \mathcal{T}_2$, they cannot have y-free sides in $\mathcal{T}_1$. This would mean that there exists a node n in $\mathcal{T}_1$ such that n is neither adjacent to, nor shares a corner with another x- (respectively y-)free node, and thus that $\mathcal{T}_1$ is not convex, which contradicts the hypothesis $\square$.

Discussion

The practical consequence of the previous results is that two strategies can be adopted when solving convex CCSPs using the 2^k-tree representation of constraints:

1. The risk of disconnection being restricted to the limit cases where the solution set is closed to the minimal resolution, one can consider as convex the connected representation with *minimal convexity deficiencies*. In the marginal cases where a disconnection occurs, it is then possible either to resort to further refinements of the quadtrees or to neglect the solution region within the disconnected area—considering that its identification requires a precision having no significance for the application. The problem will then be stated as non convex and can be solved using the search techniques we present in section 5.5.2.

2. One can only consider as convex the problems represented by convex quadtrees. We show in section 5.5.4 that non-convex quadtrees can be decomposed into convex ones and then solved by the search techniques we present in section 5.5.2. The advantage of this approach is that the risk of disconnection is precluded *a priori*, its drawback is that it may introduce an unnecessary combinatoric.

In the following, we present algorithms for checking the convexity and (x)-convexity of a quadtree.

5.4.2 Checking for (x)-convexity

In this section, we present a simple procedure called *Test-axis-convexity* deciding the (x)-convexity of a binary relation $R(x, y)$ represented by a quadtree $\mathcal{T}(x, y)$.

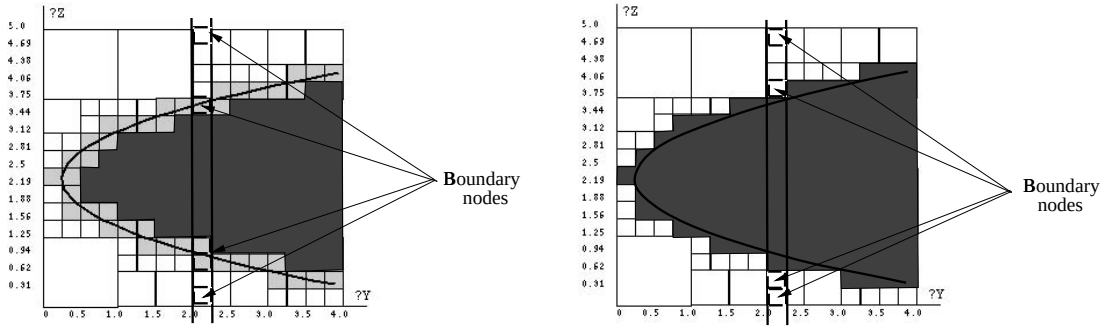


Figure 5.12: ***left:*** *boundary nodes in a conservative approximation,* ***right*** *:boundary nodes in a non-conservative approximation*

According to definition 5.10 and Lemma 5.6, a binary relation is (x)-convex if it is connected and any line parallel to the x axis cuts the solution space:

- in exactly two points when R delimits surface (inequalities),
- in exactly one point, when R delimits a point set (conjunction of inequalities and equalities)

In the case of quadtrees, a line is approximated by an interval-line whose size corresponds to the degree of precision chosen. The convexity deficiencies smaller than this precision are consequently neglected.

Moreover a quadtree approximates the effective boundaries of a feasibility space using the boundary nodes, determined either by:

- the gray nodes and the white nodes with free sides, if the representation is conservative (see Figure 5.12, left)
- or by the white nodes with at least one free side, otherwise (see Figure 5.12, right)

By extension, we will say that a quadtree $\mathcal{T}(x, y)$ is (x)-convex, if any interval-line of size ε_y (where ε_y is the smallest interval size for variable y), parallel to the x axis contains exactly:

- two sets[3] of adjacent boundary nodes, when $\mathcal{T}$ represents a surface (i.e., contains white nodes), Figure 5.13,

[3]We consider *sets* of boundary nodes, rather than a single boundary node, because a boundary point can be approximated by more than one node

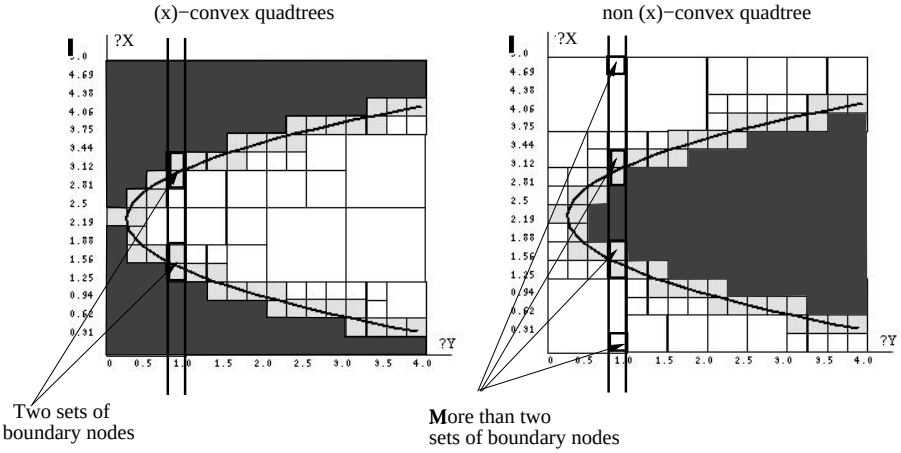


Figure 5.13: ***left:*** *(x)-convex quadtree , and* ***right :*** *non (x)-convex quadtree for an inequality*

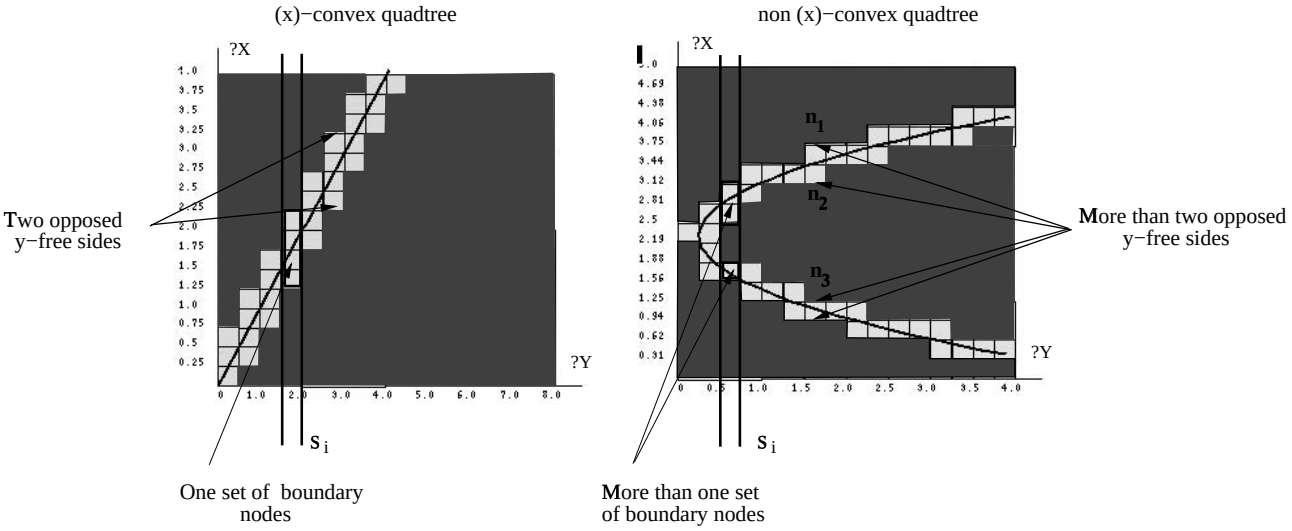


Figure 5.14: ***left:*** *(x)-convex quadtree , and* ***right :*** *non (x)-convex quadtree for an equality*

- one set of adjacent boundary nodes, when $\mathcal{T}$ represents a point set (i.e., contains only gray nodes), Figure 5.14.

The algorithm *Test-axis-convexity* simply examines the boundaries slice by slice through the y axis. Each slice corresponds to an interval-line of width ε_y. We define:

Definition 5.15 *Two nodes n_1 and n_2 taken from a slice s_i of a quadtree $\mathcal{T}(x,y)$ have opposed free sides in a given direction l, if n_1 has no upper neighbour and n_2 has no lower neighbor in the direction l.*

Test-axis-convexity is mainly based on the following observation:

Proposition 5.3 *A quadtree $\mathcal{T}(x,y)$ is (x)-convex, if any interval-line of size ε_y, parallel to the x axis, contains exactly two opposed y-free sides (see figure 5.14).*

Proof. Firstly, it is obvious that a slice s_i cannot contain only one free side (the upper and lower nodes within s_i having opposed y-free sides). Moreover, within a given slice, the feasibility nodes are necessarily delimited by a pair of y-free side nodes. Hence, the only boundary nodes relevant for testing (x)-convexity within a slice are those having y-free sides. Now suppose that s_i contains more than two free sides, this means that there exist at least three nodes n_1, n_2 and n_3 such that: *i.* n_1, n_2, n_3 have y-free sides, *ii.* the block of nodes between n_1 and n_2 is feasible and *iii.* the block of nodes between n_2 and n_3 is infeasible (otherwise n_2 would not be a y-free side node). This means that s_i cuts the boundaries at least three times and that the quadtree is not (x)-convex. □

In the algorithm of Figure 5.15, we follow the convention that the boundary nodes are stored in B with the minimum size for y. Each node of B is consequently contained within a single slice. We also assume that the boundary nodes are stored in B with a marker indicating which of their side is y-free (upper or lower). $S(i)$ denotes the slice corresponding to the sub-interval i for y. D_y is the domain size of variable y and ε_y is the minimal interval size for y in the decomposition. The algorithm *Test-axis-convexity* returns the boolean value `true` if the quadtree $\mathcal{T}(x,y)$ is (x)-convex. Two adjacent slices, $S(j-1)$ and $S(j)$, are not connected when there is no overlap between the interval over the x-axis delimited by the nodes of $S(j-1)$ and the interval over the x-axis delimited by the nodes of $S(j)$.

Test-axis-convexity$(\mathcal{T}(x,y), x)$:

```
1  {Initialization}
2  B ← Find the boundary nodes with y-free sides in T(x,y);
3  foreach n ∈ B do
4     i ← decode(∏_y(n)); {find the slice i of n}
5     S(i) ← S(i) ∪ {n};
6     if S(i) contains more than two y-free sides
7        then return Fail; fi
8  od
9  if ∪(i) ≠ D_y then return Fail; fi
10 for j = 2 to D_y/ε_y do
11    if S(j-1) is not connected to S(j) then return Fail; fi
12 od
13 return True
```

Figure 5.15: *Algorithm for testing the (x)-convexity of a quadtree*

Theorem 5.11 *The algorithm* `Test-axis-convexity` *correctly decides the (x)-convexity of a quadtree representation* $\mathcal{T}(x,y)$ *in time* $O(Nlog_4(N) + 2^{D_y/\varepsilon_y})$ *and space* $O(2^{D_y/\varepsilon_y})$, *where* N *is the number of feasibility nodes,* D_y *is the domain size of variable* y *(i.e., the interval projection of the quadtree over the* y *axis) and* ε_y *is the minimal interval size for* y *in the decomposition.*

Proof. The proof of correctness derives directly from proposition 5.3. Since each slice corresponds to the maximum precision and since the boundary nodes necessarily cover the whole feasibility space, the minimal possible (x)-convexity deficiencies are examined exhaustively. Step 2 consists of retrieving, among the set of feasible nodes, those having a y-free side. For each node, the algorithm has to check for the existence of at most two adjacent nodes. As seen in section 4.2, the code of adjacent nodes is computed in constant time and retrieving a node of a given code in a quadtree is in $O(log_4(N))$, where N is the number of feasible nodes. Step 2 is therefore in $O(Nlog_4(N))$. Since we consider that each node in B is stored with a minimal interval length for y, the number of nodes in B is in $O(2^{D_y/\varepsilon_y})$. According to proposition 5.3, each slice is visited at most three times. Steps (9-12) check for the connectivity of the quadtree. When these steps are executed, each slice contains at most two nodes (otherwise step 6 of the algorithm would have reported failure). Step 9 tests whether the slices cover the whole domain (if this is not the case the domain is disconnected). It then remains to test whether the blocks in each pair of

adjacent slices are connected to one another, which is done by steps (10-12).

The worst case occurs when the quadtree is connected and contains no (x)-convexity deficiency (each slice will then be examined). Thus, the time complexity of *Test-axis-convexity* is $O(Nlog_4(N) + 2^{D_y/\epsilon_y})$. In case the quadtree contains no (x)-convexity deficiencies, $2^{D_y/\epsilon}$ slices will be constructed and stored. The space requirements are therefore in $O(2^{D_y/\epsilon})$ □.

For a fixed number of recursive subdivisions, the time complexity of the algorithm *Test-axis-convexity* is $O(Nlog_4(N))$ where N is the number of nodes in the quadtree of maximal size. In practice, using iterative versions of the consistency algorithms presented in chapter 4 (as *Iterative-*$\mathcal{T}$-AC-3 of Figure 4.7), a small number of subdivisions (≤ 10) is generally sufficient to reach results with the required precision. This means that for practical cases we generally have $2^{D_y/\epsilon_y} << Nlog_4(N)$.

(x)-convex quadtrees with minimal convexity deficiencies

As discussed in section 5.4.1, a loose interpretation of convexity for quadtrees consists of admitting as convex the representations which are both (x) and (y)-convex and for which the convexity deficiencies are *minimal*(see Definition 3.4). This interpretation does not remove completely the risk of disconnection but limits it to marginal cases. A simple way of verifying that the convexity deficiencies in a (x) and (y)-convex quadtree are minimal consists of joining the exterior corners of the boundary nodes using line segments, and computing the slope of each line segment. This can be done in $O(c)$ time complexity where c is the number of convexity deficiencies in the quadtree.

5.4.3 Checking for quadtrees convexity

A strict characterization of convexity for quadtrees is given by definition 5.14. A simple algorithm for checking the convexity of a quadtree is given in Figure 5.16. It uses a variant of the algorithm *Test-axis-convexity* taking the boundary nodes as argument rather than the quadtree itself. D_x and D_y are respectively the domain size for the variables x and y (the domains are given by the unary projections of the quadtree over the axis x and y). Note that constructing the sets B_x and B_y can be done using a single iteration over $\mathcal{T}(x,y)$'s nodes (two steps, 2 and 3, are introduced in the algorithm *Test-convexity* for clarity only). Similarly, the slices $S(j)$ are constructed while testing (x)-convexity and

Test-convexity$(\mathcal{T}(x,y))$:

```
1  {Initialization}
2  B_y ← Find in T(x,y) the boundary nodes with y-free sides;
3  B_x ← Find in T(x,y) the boundary nodes with x-free sides;
4  S^y_{j:1,D_y/ε} ← Insert nodes of B_y in their respective slices;
5  S^x_{j:1,D_x/ε} ← Insert nodes of B_x in their respective slices;
6  if Test-axis-convexity(B_y, x) ∧ Test-axis-convexity(B_x, y)
7    then
8         for j = 2 to D_y/ε_y do
9             if ‖min-node(S^y(j−1)) − min-nodeS^y(j)‖ > D_x/ε
10               ∨
11               ‖max-node(S^y(j−1)) − max-nodeS^y(j)‖ > D_x/ε
12               then return Fail; fi
13        od
14        for j = 2 to D_x/ε_x do
15            if ‖min-node(S^x(j−1)) − min-nodeS^x(j)‖ > D_y/ε
16               ∨
17               ‖max-node(S^x(j−1)) − max-nodeS^x(j)‖ > D_y/ε
18               then returnFail; fi
19        od
20        return True.
```

Figure 5.16: *Algorithm for testing the convexity of a quadtree*

do not need, in principle, to be reconstructed in steps 4 and 5. The functions *max-node* and *min-node* respectively return the upper x coordinate of the upper node in a slice (upper meaning with the higher x coordinate) and the lower x coordinate of the lower node in a slice. The quadtree $\mathcal{T}(x,y)$ is declared convex if the size difference between two consecutive blocks of nodes (slices) is of at most one node on either side.

Theorem 5.12 *The algorithm* Test-convexity *correctly decides the convexity of a quadtree $\mathcal{T}(x,y)$ in time $O(2.Nlog_4(N)+2^{l+1})$ and space $O(2^{l+1})$, where N is the number of feasibility nodes and l is the number of levels in the quadtree($l = D_y/\varepsilon_y = D_x/\varepsilon_x$)*

Proof. Once the quadtree has been checked for (x) and (y)-convexity, it remains to verify that the maximal indentations along the boundaries have the size of a single minimal node. Since the algorithm iterates over all the possible slices of minimal size, in both dimensions x and y, it is guaranteed that all possible convexity deficiencies are checked for. Each (x)-convexity test of steps 4 and 5 is in $O(Nlog_4(N)+2^l)$ and each set of iterations of steps 8-13 an 14-19 is in $O(2^l)$. The worst case complexity occurs when the quadtree is convex (each slice is visited in both dimensions). Thus, the time complexity of test-convexity is in $O(2(Nlog_4(N)+2^l)) = O(2Nlog_4(N)+2^{l+1})$. In case where the quadtree is convex $2*2^l$ slices will be constructed and stored. The space requirements are therefore in $O(2^{l+1})$ □.

5.4.4 Checking for (x,y)-convexity

Since we translate n-ary constraints into ternary ones for tractability purposes, the procedure presented earlier need only to be generalized to 3-dimensions.

We recall that a relation is (x,y)-convex if its projection over any plane $(x \times y)$ (z varying) yields a convex binary region. In our discretized representation, a plane $(x \times y)$ is approximated using an interval-plane of width ε_z (where ε_z is the minimal interval for the variable z) (see Figure 5.17).

The definition of a (x,y)-convex relation naturally translates as follows to the case of octrees:

Definition 5.16 *An octree $\mathcal{T}(x,y,z)$ is (x,y)-convex , if the set of $\mathcal{T}(x,y,z)$'s nodes, taken from any $(x \times y)$ interval-plane of size ε_z, yields a convex quadtree $\mathcal{T}(x,y)$ when projected over the plane $(x \times y)$.*

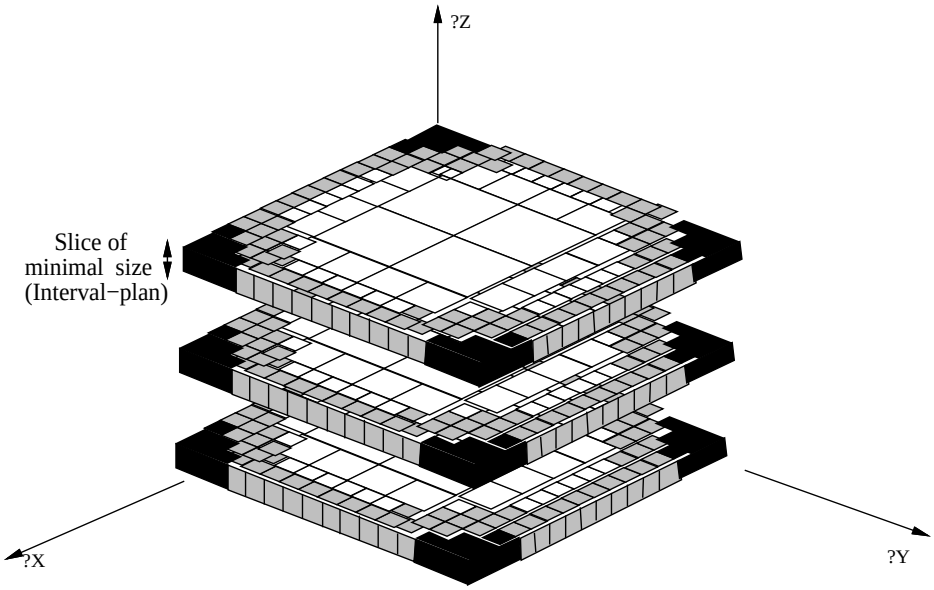


Figure 5.17: *A (x,y)-convex octree can be sliced into a set of convex (x,y)-quadtrees*

Test-plane-convexity$(\mathcal{T}(x,y,z),(x,y))$:

1 $\{Initialization\}$
2 **foreach** $n \in \mathcal{T}(x,y,z)$ **do**
3 $i \leftarrow decode(\prod_z(n))$; $\{$*find the z slice i of n*$\}$
4 $S(i) \leftarrow S(i) \bigcup \{n\}$; **od**
5 **if** $\bigcup(i) \neq D_z$ **then** *return***Fail**; **fi**
7 **for** $j = 1$ **to** D_z/ε_z **do**
8 **if** $(j > 1 \wedge S(j-1)$ *is not connected to* $S(j))$
9 **then** *return* $Fail$;
10 **else if** $\prod_{(x,y)}(S(j))$ *is not a convex quadtree* **then** *return* $Fail$; **fi**
11 **od***return*$True$

Figure 5.18: *Algorithm for testing the (x,y)-convexity of an octree*

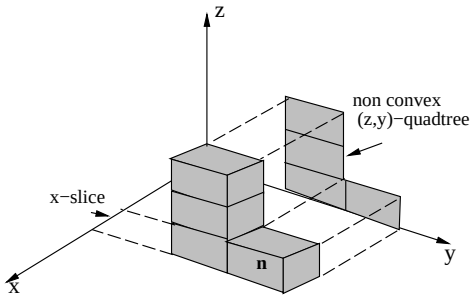


Figure 5.19: *In an octree, a non-minimal (x,y)-convexity deficiency is incompatible with (y,z)-convexity*

The algorithm *Test-plane-convexity* of Figure 5.18 simply examines iteratively the slices over the z axis, each slice of size ε_z being separately checked for convexity. Steps 5-11 verify the simple connectivity of the octree. Two adjacent slices, $S(j-1)$ and $S(j)$ are not connected when there is no overlap between the interval over the z-axis delimited by the nodes of $S(j-1)$ and the interval over the z-axis delimited by the nodes of $S(j)$. The convexity test of Step 10 can be done either according to the loose or strict definition of convexity for quadtrees. The complexity measures we give below assume that the strict definition is adopted. The worst case occurs when the octree is connected and (x,y)-convex (each slice will then be examined). An overestimation of the number of nodes per slices is given by N, where N is the number of feasibility nodes in the octree. The worst case complexity can therefore be estimated at $O(N + 2^l.(2N.log_4(N) + 2^{l+1})) = O(N + 2^{l+1}(N.log_4(N) + 2^l)$, where N s the number of nodes in the octree and l is the number of levels in the octree. Hence, for a fixed precision, the convexity of checking (x,y)-convexity is in $O(N + Nlog_4(N))$.

5.4.5 Checking for octree convexity

Octree convexity

The strict definition of convexity for quadtrees (see section 5.4.1) was introduced with the purpose of removing the risk of disconnection due to intersection. We have seen that even if the quadtrees are both (y) and (x)-convex, their intersection can yield disconnected regions when the size of convexity deficiencies exceeds the size of a minimal node.

Similarly to the case of quadtrees we now define a strict definition of convexity for octrees. We first show that in the case of octrees, imposing the (x,y), (x,z) and (y,z)-

convexity directly restricts the size of convexity deficiencies to the size of minimal nodes and hence, precludes the risk of disconnection.

Proposition 5.4 *If an octree $\mathcal{T}(x,y,z)$ (x,y), (x,z) and (y,z)-convex, any of its (x,y) (respectively (x,z),(y,z)) -free node:*

- *is either adjacent to another (x,y) (respectively (x,z),(y,z)) -free node,*
- *or shares a corner with another (x,y) (respectively (x,z),(y,z)) -free node*

Proof. Suppose that there exists a node (x,y) n in $\mathcal{T}(x,y,z)$ being neither adjacent to another (x,y)-free node, nor sharing a corner with another (x,y)-free node. This would mean that the x-slice containing n and its neighbors does not yield a convex quadtree when projected over the $(y \times z)$ plane (see Figure 5.19). This contradicts the fact that $\mathcal{T}(x,y,z)$ is (y,z)-convex (considering the other planes leads to symmetric cases) □.

We will consequently define a convex octree as follows:

Definition 5.17 *An octree $\mathcal{T}(x,y,z)$ is convex if and only if it is (x,y)-, (x,z)- and (y,z)-convex*

Proposition 5.5 *Convexity is closed under octree intersection*

Proof. Let T_1 and T_2 be two convex octrees. Suppose that $T_1 \oplus T_2$ is not convex. This would mean that there exists a z-slice $S_z(i)$ in $T_1 \oplus T_2$ such that the projection of $S_z(i)$ over the plane $(x \times y)$ does not yield a convex quadtree. We denote as P the projection of $S_z(i)$ over the plane $(x \times y)$. The slice $S_z(i)$ is obtained by intersecting the corresponding slices $S_z^1(i)$ and $S_z^2(i)$ in T_1 and T_2. Both $S_z^1(i)$ and $S_z^2(i)$ slices have convex projections P_1 and P_2 over the plane $(x \times y)$ (T_1 and T_2 being convex). Moreover, the intersection of P_1 and P_2 is known to be convex (proposition 5.2). Since each slice is constituted of minimal size nodes, we have $P = P_1 \oplus P_2$, P being non convex and $P_1 \oplus P_2$ being convex. Contradiction □.

Therefore, checking for the convexity of an octree consists simply of testing whether it is (x,y)-, (x,z)- and (y,z)-convex. Each test being in $O(N + 2^{l+1}(N.log_4(N) + 2^l)$, the whole complexity is in $O(N + 2^{l+1}(N.log_4(N) + 2^l)$.

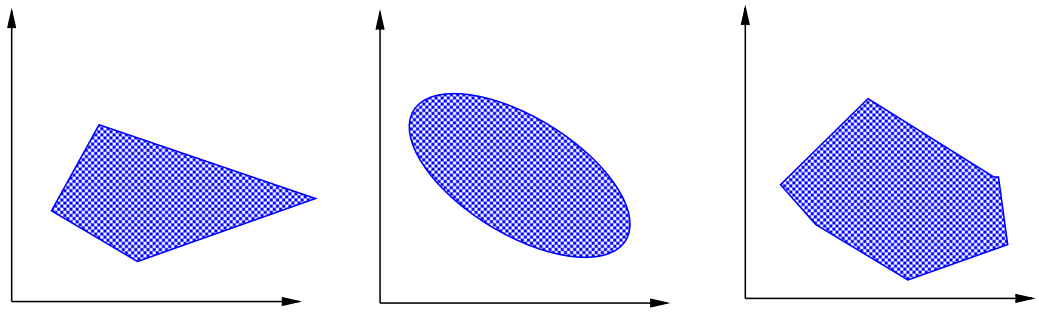


Figure 5.20: *Each relation of a convex CCSP determines a convex solution space. Convex CCSPs admit polynomial time solutions.*

5.5 A classification of continuous CSPs

A general CCSP may admit no convex constraint network representation. Moreover, even if the initial problem has interesting partial convexity properties ((x)-convexity, (x,y)-convexity), consistency algorithms may not preserve these properties since intersecting two non convex regions may results in an arbitrary number of distinct sub-regions. For these reasons, the results of Theorems 5.7 and 5.9 are valid only on the *a posteriori* networks computed by consistency algorithms. Concerning the *a priori* networks, similar results can be formulated only for the classes of problems where (x)-convexity (or its generalization to higher dimensions) is closed under composition and intersection of constraints. In order to bound a priori the complexity of solving CCSPs, it is desirable to identify the classes of problems for which consistency algorithms preserve interesting convexity properties.

We can distinguish three classes of CCSPs, as described in the following.

5.5.1 Convex CCSPs

Definition 5.18 : ***Convex CCSPs***
A CCSPs,(V, C, D)*, is* convex *if and only if every relation of* C *is convex*

The intersection of two convex regions is necessarily convex. For the case of *convex* CCSPs (see Figure 5.20), the a posteriori network computed by consistency algorithms is therefore guaranteed to be convex. This category of CCSPs consequently admit polynomial time solutions (path-consistency using quadtrees is in $O(n^3)$, as shown in chapter 4). They can be considered as the generalized counterparts of simple temporal problems [38].

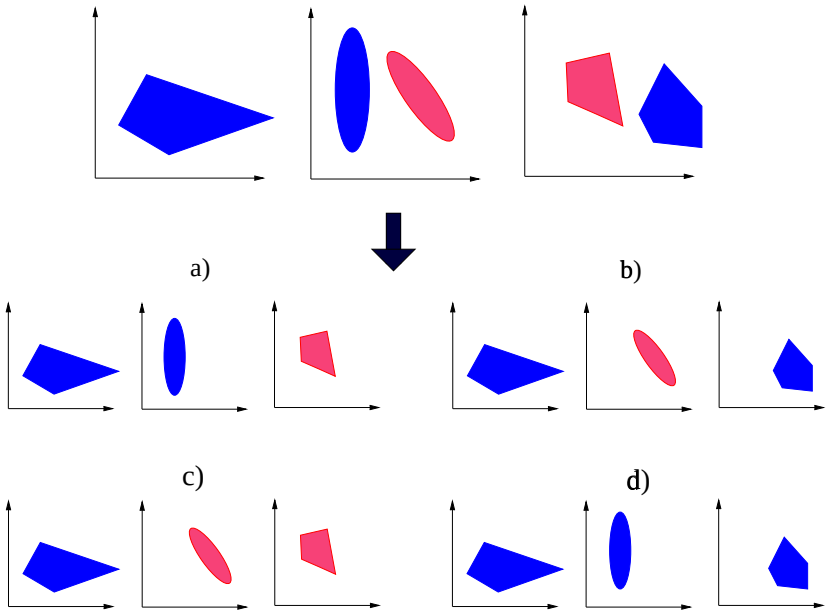


Figure 5.21: *Multi-convex CCSPs can be decomposed into convex sub-problems, one for each possible combination of convex sub-region. In this example, the original CCSP can be solved by determining the solutions of the convex CCSPs (a), (b), (c) and (d) (sub-problems).*

5.5.2 Multiconvex CCSPs

Definition 5.19 *: **Multi-convex CCSPs***

A CCSP, (V, C, D)*, is* multi-convex *if and only if:*

i. There exist non path-connected relations in C*,*

ii. Each path-connected relation of C *is also convex,*

iii. The solution space of each non path-connected relation of C *is constituted by a set of convex regions,*

We can prove that solving a multi-convex CCSP is NP-hard. The following Theorem is an extension of Davis Theorem on temporal CSPs (TCSPs) [38].

Theorem 5.13 *Deciding the consistency for a multiconvex CCSP is NP-Hard in the number of disconnected regions*

Proof. Similarly to the case of TCSPs, the proof is done by reduction from 3-coloring. Let $G = (V, E)$ a graph to be colored. Let C be a multiconvex binary CCSP constructed as follows: each variable X_i takes its values within three convex intervals I_1, I_2 and I_3

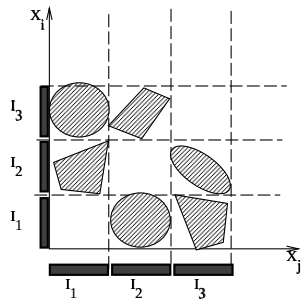


Figure 5.22: *Solving a multi-convex CCSP is NP-Hard*

standing for three admissible colors. The relation associated with each pair of variables is depicted in Figure 5.22: admissible pairs of values belonging to three disconnected convex regions. The convex regions specify the fact that if variable X_i takes a value from interval I_k then variable X_j takes its value from $I_{k'}$ so that $k \neq k'$. The relations of the multi-convex CCSPs consequently restrict two variables X_i and X_j to have different colors (i.e., take their values within different intervals). Hence, C is consistent if and and only if G is 3-colorable.

A multi-convex CCSP can be decomposed into convex sub-problems, one for each possible combination of convex sub-regions (see Figure 5.21). Each sub-problem is a convex CCSP. Solutions to multi-convex CCSPs can be determined by solving individually each convex subproblem and then combining their solutions, as stated in the following Theorem (also derived from results on temporal CSPs [38]):

Theorem 5.14 *The minimal network $\mathcal{N}_{min}$ of a multi-convex CCSP, $\mathcal{P}$, satisfies $= \mathcal{N}_{min} \bigcup \mathcal{N}^c_{min}$ where $\mathcal{N}^c_{min}$ is the minimal network of a convex subproblem c and the union is over all the possible combination of convex sub-problems.*

Proof. The demonstration is analog to the one presented for TCSPs in [38]. The problem defined by $\bigcup \mathcal{N}^c_{min}$ has the same solution space as $\mathcal{P}$. Since $\mathcal{N}_{min}$ is by definition the tightest network equivalent to $\mathcal{P}$, we have $\mathcal{N}_{min} \subseteq \bigcup \mathcal{N}^c_{min}$. Supposing that $\mathcal{N}_{min}$ is strictly tighter than $\bigcup \mathcal{N}^c_{min}$ means that there exists a convex subproblem $\mathcal{N}^{c'}_{min}$ and a point $p = (v_1, ..v_k, ..v_n)$ with $v_k \in D_k$ such that $p \in \mathcal{N}^{c'}_{min}$ and $p\neg \in \mathcal{N}_{min}$. p is a solution of the subproblem $\mathcal{N}^{c'}_{min}$ and is consequently also a solution of $\mathcal{P}$. Hence, $p \in \mathcal{N}_{min}$ which contradicts the hypothesis. The complexity of solving a multi-convex CCSP is exponential in the number of convex subproblems. The worst case time complexity is given by $O(n^3 d^r)$

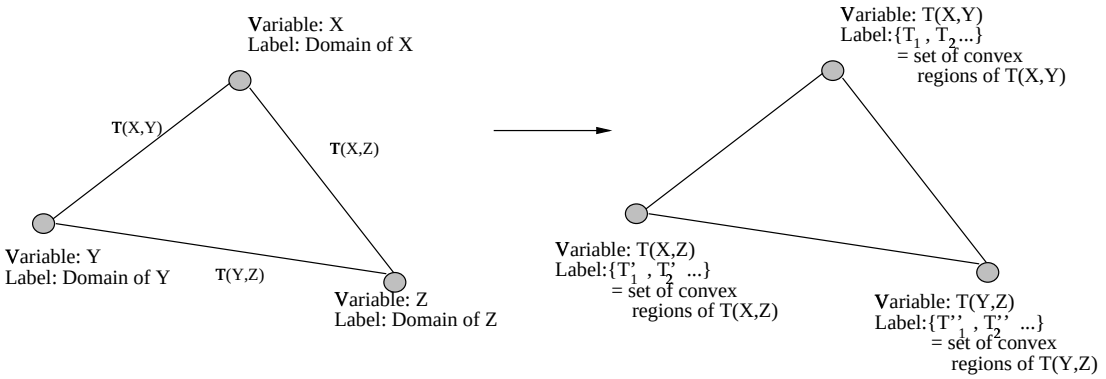


Figure 5.23: *Dual discrete representation (**right**) for a multi-convex CCSP (**left**)*

for a binary network and by $O(n^5 d^r)$ for a ternary one, where n is the number of variables, d the number of disconnected convex subproblems and r the number of constraints in the network. $O(n^3)$ for binary CCSPs (respectively $O(n^5)$ for ternary ones) is the complexity of solving an individual convex subproblem as stated in chapter 6.

Search

If the task consists of finding a region (or some regions) of the feasibility space rather than the whole solution space, backtracking algorithms can be proposed for multi-convex CCSPs, as alternatives to the brute force enumeration process.

Figure 5.24 gives an adaptation of the backtracking algorithm designed for TCSPs to the case of multi-convex CCSPs. In this algorithm, a multi-convex CCSP, $\mathcal{P}$, is translated into a discrete CSP where the variables correspond to the edges of $\mathcal{P}$ and the values for the variables are the set of convex regions for each edge (see Figure 5.23). The assignment $(X_1 = R_1, ..X_i = R_i)$ is consistent if and only if the corresponding convex CCSP is consistent.

The algorithm in Figure 5.24 is defined by two recursive procedures *Forward* and *Go-back*. The former extends a current partial assignment, when possible, while the latter backtracks on failure. The procedure *Solve-convex-CSP* returns solutions of the subproblem defined by the convex regions $\{R_1, \ldots, R_m\}$. The procedure `consistent` determines if the partial sub-problem defined by the convex regions $\{R_1, \ldots, R_i, R_j\}$ is consistent (using a path-consistency algorithm for binary problems and a (3,2)-relational-consistency algorithm for ternary problems).

Forward(R_1, ... R_i):

```
1  if i = m then
2                  S ← S ⋃ Solve-convex-CSP(R_1, ... R_m);
3                  Go-back(R_1, ... R_m);
4                  C_{i+1} ← ∅;
5                  foreach R_j ∈ D_{i+1} do
6                    if consistent(R_1 ... R_i, R_j) then
7                                                  C_{i+1} ← C_{i+1} ⋃{R_j} fi od
8                  if C_{i+1} ≠ ∅ then
9                                    R_{i+1} ← popC_{i+1};
10                                   removeR_{i+1}fromC_{i+1};
11                                   Forward(R_1 ... R_i, R_{i+1});
12                               else Go-back(R_1, ... R_i); fi
```

Go-back(R_1, ..., R_i):

```
1  if i = 0 then returnFail
2           else if C_i ≠ ∅ then
3                                  I_i ← popC_i;
4                                  removeR_ifromC_i;
5                                  Forward(R_1 ... R_i)
6                              else Go-back(R_1, ... R_{i-1});
```

Figure 5.24: *A backtracking algorithm for multi-convex CCSPs*

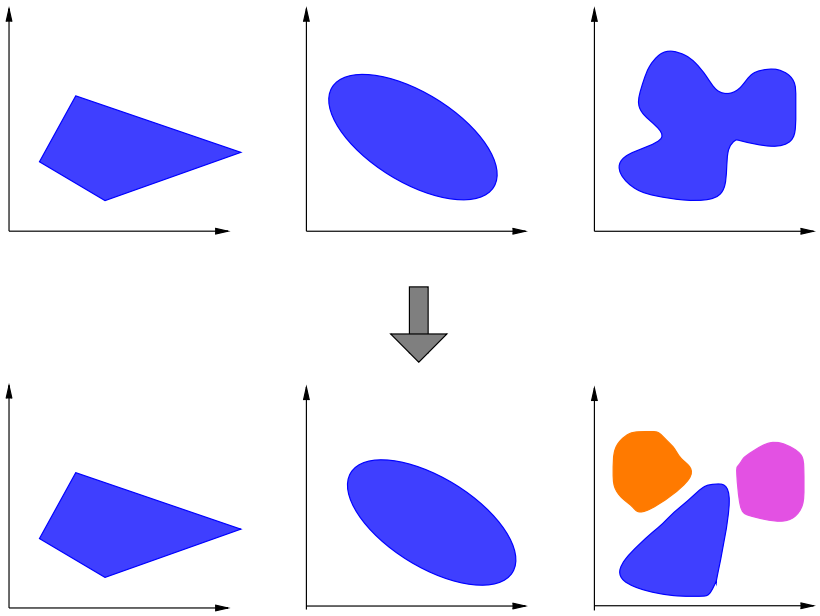


Figure 5.25: *The resolution complexity is difficult to estimate a priori for non-convex CCSPs. Non-convex CCSPs can be decomposed into multi-convex ones in some particular cases.*

Since the complexity of solving the convex subproblems is bounded (and polynomial), more sophisticated search schemes can be designed that incorporate local consistency checking or various heuristic enhancements (variable ordering, directed-backtracking, backjumping etc...) [135].

5.5.3 Non convex CCSPs

Definition 5.20 *: Non convex CCSPs*

A CCSP, (V, C, D)*, is* non-convex *it there exist non convex relations in* C*.*

The resolution of non convex CCSPs might be plagued by the *splitting* problem, first described for interval-based methods [34, 75]. This problem occurs when the revision of a given constraint results in disjoint intervals, which splits the original problem into several sub-problems and generates a combinatorial effect. Hyvönen discusses this issue extensively in [75].

From our perspective, the splitting problems is related to the fact that intersecting non convex regions may result in an undetermined number of disjoint subregions. These subregions, if non convex, can in turn split again during the successive constraint revisions by the consistency algorithms. If these splittings have no direct influence on the

consistency techniques themselves, they clearly introduce a combinatorial effect in the subsequent search for a solution.

The complexity of solving non-convex CCSPs is therefore difficult to estimate a priori. In the best case, the consistency algorithm used (path for binary and (3,2)-relational-consistency for ternary problems) may create an (x)-convex (respectively (x,y)-convex) CCSP from a set of non-convex relations. This results in an a posteriori polynomial time resolution since the subsequent search process is linear in time (backtrack-free instantiation).

In the worst case however, the revision of each non convex relation may create several non-convex sub-relations, which can in turn split again. Of course, if we disregard the issue of fractal boundaries, one can assume that the combinatorial explosion is limited by the number of convexity deficiencies existing in the relations. From a practical point of view, representing constraints with their 2^k-tree approximations also limits the combinatorial explosion, since convexity deficiencies smaller than the maximum precision are neglected.

In the following section we discuss the possibility of translating non convex CCSPs into multi-convex ones (Figure 5.25). Such a recasting of non convex problems is desirable since it would enable:

- the prediction, a priori, of the resolution complexity for general CCSPs
- the utilization of search procedures such as those described for multi-convex CCSPs

5.5.4 Decomposing a non-convex relation

In principle, there exists effective solution spaces, admitting no multi-convex decompositions — for example the solution space delimited by a torus in 3D. This is not the case in practice when using the 2^k-tree discretized representation of constraints. A simple decomposition would in effect consist of considering each individual 2^k-tree node as a convex sub-problem. The decomposition algorithm we propose simply tries to improve this trivial decomposition technique by assembling as much nodes as possible in convex sets.

Binary relations The procedure *decompose* in Figure 5.26 is based on the same "slicing" principle used for testing the convexity properties of quadtrees and octrees. For

Decompose$\mathcal{T}(x,y)$):

```
1  begin
2    Convex-in-x ← decompose-over-axis(T(x,y),y);
3    foreach region ∈ Convex-in-x do
4      Convex-sets ← Convex-sets ⋃ decompose-over-axis(region,x);
5    end
```

Decompose-over-axis($\mathcal{T}(x,y),d$):

```
1  begin
2    Slices ← construct the slices over the d axis
3    return assemble-convex-blocks(Slices,d)
4  end
```

Assemble-convex-blocks($Slices,d$):

```
1  begin
2    Convex-sets ← ∅;
3    k ← 1
4    repeat
5    b ← leftmost-block(Slices);
6    i ← slice of b
7    remove b from Slices(i);
8    Convex-set_k ← {b};
9    for j = next-slice(i) to D_d/ε_d do
10       b' ← find-convex-neighbor of b in S(j);
11       if b' then Convex-set_k ← Convex-set_k ⋃ {b'};
12                  b ← b';
13            else Convex-sets ← Convex-sets ⋃ Convex-set_k
14                  k ← k + 1;
15                  Exit; fi od
16   until Slices = ∅
17 end
```

Figure 5.26: *An algorithm for decomposing a non-convex quadtree into a set of convex sub-quadtrees*

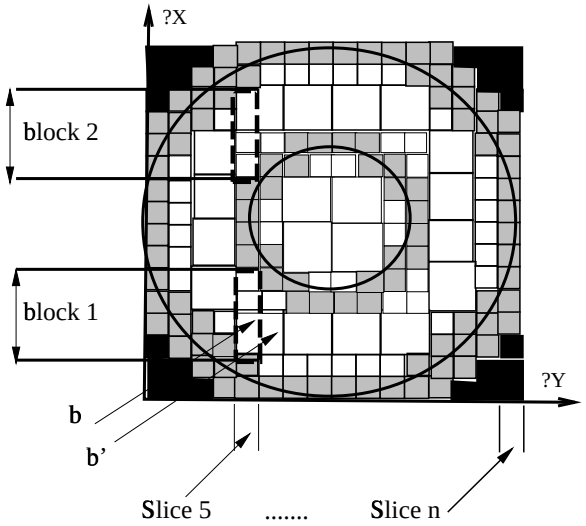


Figure 5.27: *Decomposition of a non convex quadtree into convex subquadtrees — some elements of the algorithm*

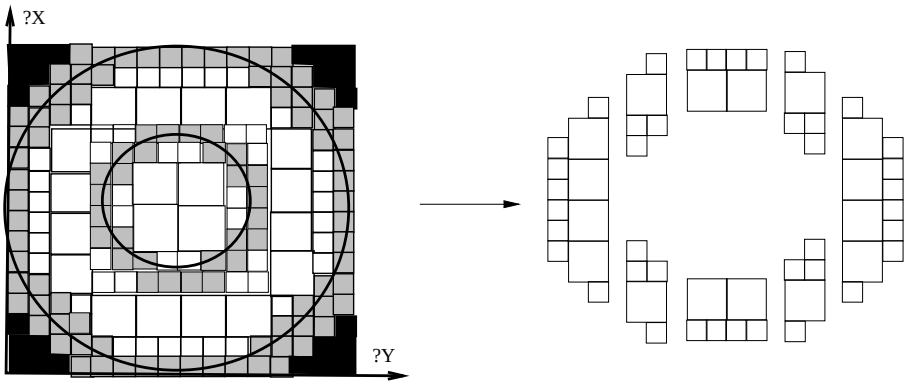


Figure 5.28: *A non convex quadtree (**left**) decomposed into eight convex sub-quadtrees (**right**)*

a binary quadtree $\mathcal{T}(x,y)$, the decomposition is carried out in both direction x and y. Constructing a convex decomposition in a given direction x requires the construction of $2^{D_y/e_y}$ slices ($O(Nlog_4N)$) (see Figure 5.27), similarly to the case of algorithm *Test-axis-convexity*. Each slice may contain more than several disjoint blocks of feasible nodes (the fifth y-slice in the quadtree of Figure 5.27 contains two disjoint blocks: block1 and block2). In the procedure `Decompose-over-axis`(see Figure 5.26), the algorithm uses the procedure `Assemble-convex-blocks` which tries to assemble a block b with a block b' (see Figure 5.27) of the adjacent slice. If a block b' is found in the adjacent slice that can form a convex set when assembled with b, b and b' are assembled and the same process is

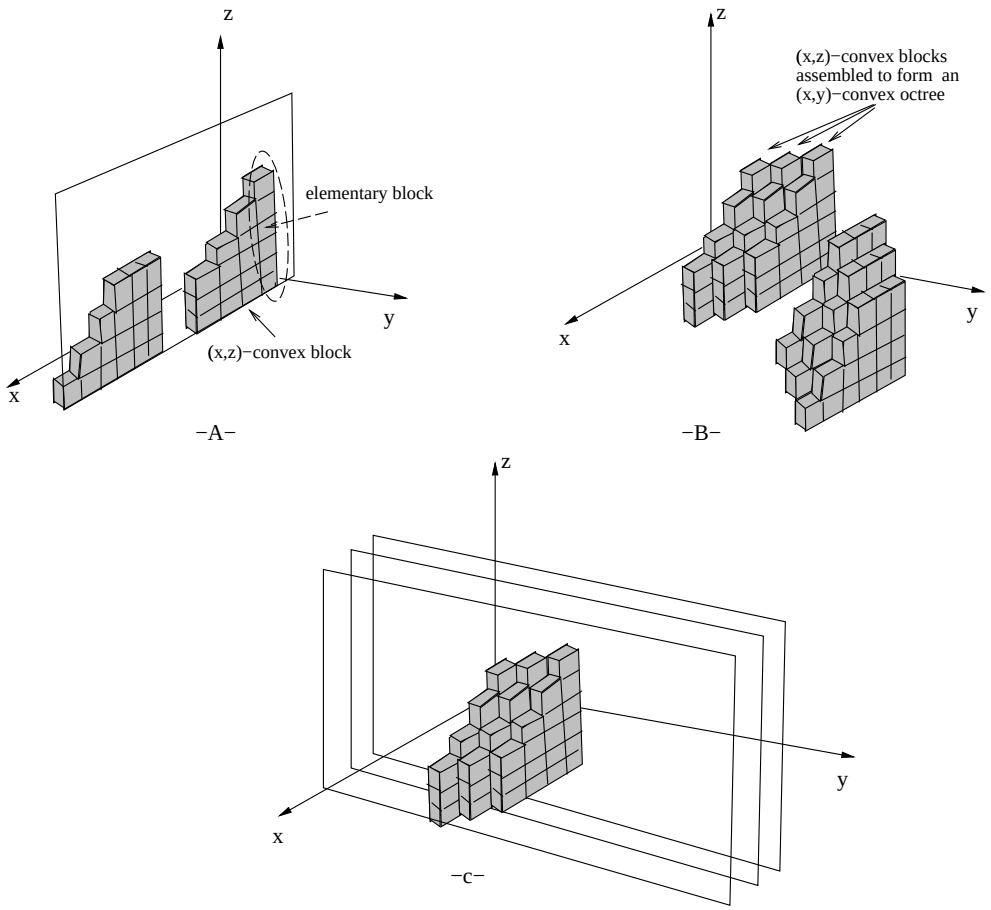


Figure 5.29: *Decomposing a non convex octree into a set of convex octrees— steps of the procedure.*

iterated with b' in place of b. If not, this means that a convex block has been completed. It is stored in the set $Convex\text{-}set_k$ of convex blocks and the whole process restart from the leftmost block until no block remains in the slices. A block can be assembled to an adjacent one if they form a convex sub-quadtree. This process results in a set of regions (group of blocks), verifying the convexity property in the x direction only. Each region has then to be decomposed into convex subregions in the y direction according to the same principle. Figure 5.28 shows the decomposition obtained for the quadtree in Figure 5.27.

Ternary relations The decomposition of octrees into convex sub-obtrees is done according to the same principle. Informally, this decomposition is carried out by the following steps:

1. D_y/ε_y two dimensional (x, z)-slices are constructed.

2. The elementary blocks within each slices are then assembled to form (x, z)-convex sets (see Figure 5.29, -A-).

3. The consecutive (x, z)-convex sets are then assembled over the y axis to form (x, y)-convex sub-octrees (see Figure 5.29, -B-).

4. Finally each sub-octree obtained is examined slice by slice over the x axis. Two sub-octrees from two consecutive slices are assembled if they form a (y, z)-convex sub-octree (see Figure 5.29, -C-).

At the end of this process, each suboctree generated is (x,y), (x,z) and (y,z)-convex. Figure 5.30 gives an example of convex decomposition for a non convex ternary relation.

5.6 Discussion and summary

In this chapter, we addressed the issue of computing a globally consistent labeling for continuous constraint satisfaction systems — a problem known to be NP-hard in its most general form. Our contributions to the issue are listed and discussed below:

1. ***Convexity as a simplifying property:*** the idea that convexity simplifies the resolution of numerical constraint systems is not new. In fact, it has been exploited since decades in linear and non linear programming methods, with the purpose of identifying efficiently a single point solution — possibly optimal with respect to some optimization criterion. Surprisingly, when it comes to the issue of computing a compact description of the *whole solution set* of a numerical constraint system, little work has been done to explicitly determine a relation between convexity and complexity.

   One of our main contributions in this chapter is to exhibit such a relation for general CSPs with continuous domains. By showing that a low degree of consistency is sufficient to compute globally consistent labeling for convex continuous problems, we establish that convexity is also of interest for consistency techniques. These results generalize those obtained by Dechter, Meiri and Pearl [38] for a restricted class of continuous problems — simple temporal problems — and can be viewed as an extension to non discrete domains of the work of van Beek on row-convexity [137].

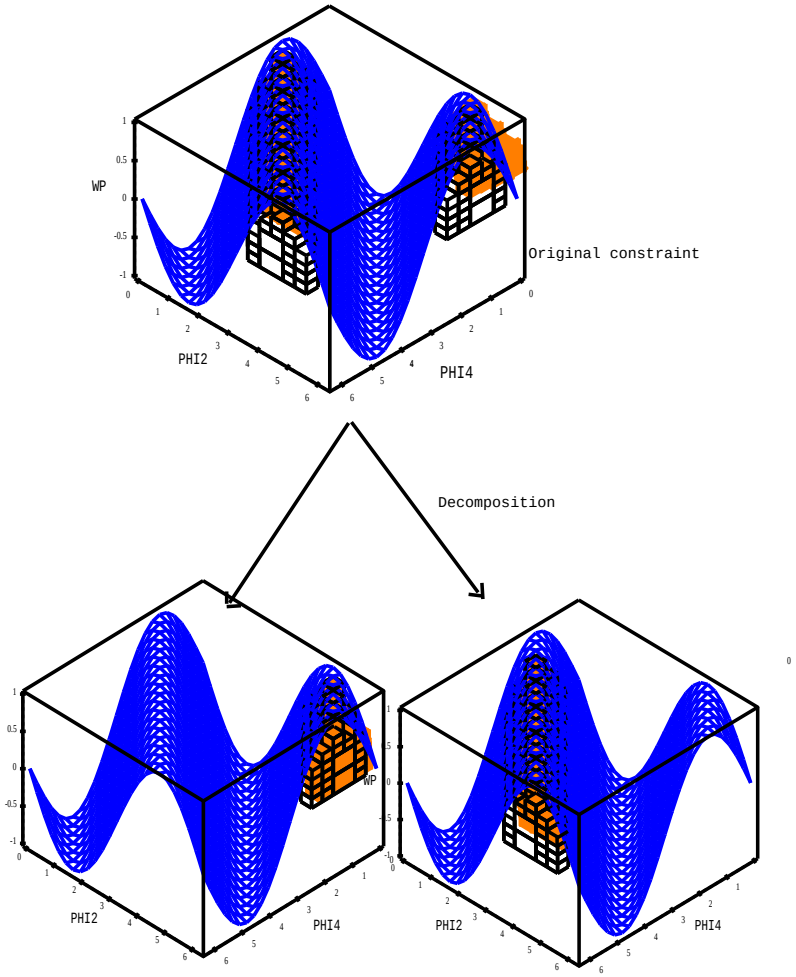


Figure 5.30: *A convex decomposition of the octree representing the ternary relation* $W_p < \cos(\Phi_2 - \Phi_4)$

2. ***N-ary convex CCSPs are tractable:*** Similarly to the case of STPs and row-convex discrete problems[4], we prove that path-consistency computes the minimal and decomposable network representation of a convex *binary* CCSP in polynomial time ($O(n^3)$). In the case of row-convex problems, van Beek and Dechter [139] proposed a generalization to non binary constraints which might pose complexity problems. In effect, the notion of relational path-consistency, required for reaching global consistency, can be difficult to enforce tractably. We contribute to this issue by showing that using an alternative generalization approach, it is possible to overcome this difficulty in continuous domains. Non binary constraints are decomposed conservatively into ternary ones. The concept of (3,2)-relational consistency is then proposed that leads to a polynomial time algorithm ($O(n^5)$) for enforcing global consistency on ternary networks. The generalization of $(3, 2)$-relational consistency to constraints of arbitrary arities is also discussed. It is interesting to note that convexity properties are easier to exploit in continuous domains than in discrete ones, since their value domains are totally ordered. In discrete problems, favorable convexity conditions may be hidden and the proper ordering of domain values exhibiting them must be searched for.

3. ***Partial convexity properties can also be useful:*** Convexity being a rather strong condition, an interesting point to study is whether weaker simplifying properties exist that reduce the complexity of solving CCSPs. We contribute to this issue by identifying a partial convexity property, called $(x_1, \ldots x_k)$-convexity, useful for deriving solutions backtrack-free. The $(x_1, \ldots x_k)$-convexity property imposes convexity conditions on sub-projections of the domain rather than on the domain itself — in the case of binary networks, it amounts to imposing convexity conditions on the unary projection of the domains. When the network computed by path-consistency (respectively (3,2)-relational consistency) verifies the (x)-convexity (respectively the (x,y)-convexity) property, we show that the network is guaranteed to be globally consistent. This result applies to a large class of problems with simply-connected solution spaces.

[4]provided that the operations on constraints conserves the row-convexity property

4. ***Checking for convexity:*** Another issue of practical interest is the question of how to check for convexity (and $(x_1, \ldots x_k)$-convexity) in numerical constraints. Using mathematical descriptions of constraints, this question is extremely complex to address in the most general case. However, the discretized decomposition of the solution space into 2^k-trees simplifies the problem by imposing a bound on the minimal size of convexity deficiencies recognized. We propose simple algorithms for testing the convexity properties of quadtrees and octrees (i.e., for binary and ternary constraints) which are sufficient for handling problems of arbitrary arities.

5. ***Tractability in terms of topological properties:*** traditional interval-based approaches refine the bounds of variable domains using numerical iterations. Their complexity is consequently difficult to estimate since it depends mainly on the number of iterations to convergence, which is hard to predict. We show in this chapter that the tractability of solving a particular CCSP is better captured by knowledge about its solution space's topological properties: *i.* CCSPs with convex solution spaces are shown to be easy (polynomial) instances, *ii.* CCSPs with multi-convex solution spaces are shown to be NP-hard in the number of disjoint sub-regions of the solution space and finally *iii.* non-convex CCSPs are of exponential complexity in the number of convexity deficiencies. Since multi-convex CCSPs can be decomposed into sets of convex sub-problems, we also show that search schemes within the space of possible convex sub-problems can be proposed that offer much flexibility— as a possibility of integrating heuristic based enhancements.

6. ***Divide and conquer, again:*** finally, we discuss the possibility of recasting non-convex problems into multi-convex ones. A simple algorithm for decomposing non-convex relations into multi-convex ones is proposed for that matter . Such a recasting allows for predicting a priori the complexity of solving non-convex CCSPs and provides the possibility of searching for solutions within convex subparts of the solution space.

Chapter 6

Implementation - Experimental Results

6.1 Implementation

A prototype implementation of the techniques and methods described in the previous chapter has been realized. It is encoded in Common Lisp (Allegro common lisp and CLIM) and includes:

- the automated construction of quadtrees and octrees,
- the consistency algorithms described in chapter 4,
  - arc, path and k-consistency algorithms,
  - a brute force version of the (3,2)-relational consistency algorithm of chapter 5,
- an instantiation procedure and the convexity checking tests described in chapter 5,
- an algorithm for decomposing non-convex problems.

Some visualization tools for quadtrees have been developed in CLIM (Common Lisp Interface Manager). The display of ternary constraints is done using MAPLE [17] facilities.

The rest of this chapter illustrates how these tools are used for computing the globally consistent labeling of some example of CCSPs.

6.2 A simple binary example

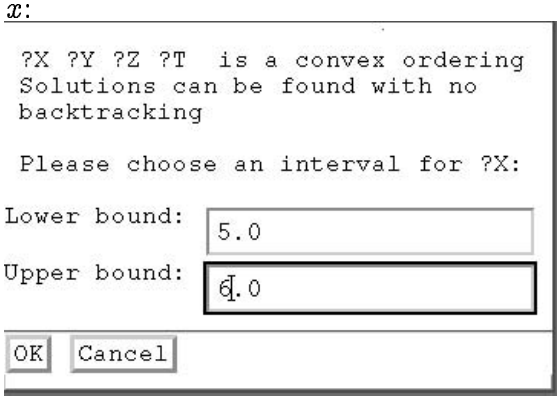
This section illustrates on a simple example the resolution method we propose for binary continuous problems. It shows how path consistency allows for the computation of globally consistent solutions when certain partial convexity properties are verified.

The problem consists of a set $\mathcal{S}$ of 19 equations involving 4 variables x,y ,z, and t:

$$\mathcal{S} = \begin{array}{ll} x + y < 8 & y - x > 0 \\ y - \frac{1}{(x-4)^2} > 0 & y - (Z - 2.3)^2 > 0.25 \\ y + z^2 < 3.75 & z - e^{(4-x)} < 2 \\ z - e^{(X-4)} < 1.5 & z - 0.5x > -2 \\ z + 1.25x > 1.25 & z - 0.075x > 0 \\ y - 0.334t < 2 & y - 1.5t < 1 \\ y + 0.2t > 1 & y - 0.6t > -3 \\ y + 0.25t < 5.5 & t - (z - 2.3)^2 > 0.25 \\ t + (z - 2)^2 < 3.75 & t - e^{(4-X)} < 2 \\ t - e^{(X-4)} < 1.5 & \end{array}$$

Given a certain degree of precision (here 5 levels of decomposition), the system implemented starts by computing the quadtree representation for the set $\mathcal{S}$ of constraints. Figure 6.1 shows the set of quadtrees constructed. Note that after this operation, the number of constraints reduces to six *total* constraints. The CCSP of Figure 6.1 is not convex: the constraint between x and y yields two disconnected regions and the other constraints have convexity deficiencies whose sizes exceed the size of a minimal node.

After enforcing path consistency(algorithm $\mathcal{T}$-PC2, section 4.3), the solution regions obtained are still non-convex (see Figure 6.2) . However, the algorithm *Find-order* (section 5.3.1) identifies (x, y, z, t) as an (x)-convex ordering. According to theorem 5.8, the solutions to the problem can be derived backtrack-free. The user is asked to enter an initial interval-value for x:



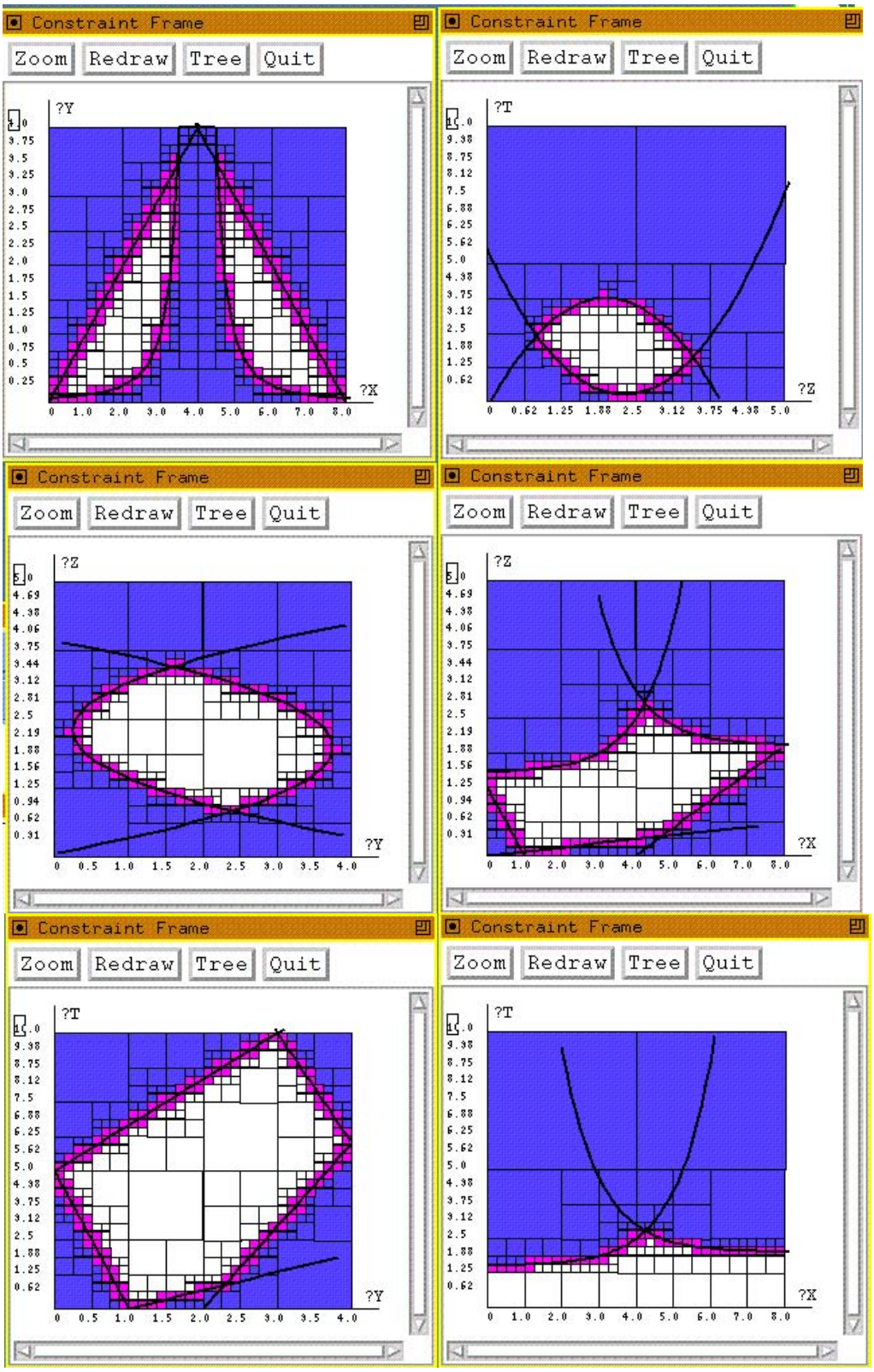


Figure 6.1: *Quadtrees for a simple binary example*

and the backtrack free instantiation process can start. Figure 6.3 shows one of the interval solutions derived by this process.

The prototype implementation on a Silicon Graphics Iris 4D using Allegro Common Lisp 4.2 gives the following statistics:

- Construction and display of quadtrees:

```
; cpu time (non-gc) 19,270 msec user, 560 msec system
; cpu time (gc)     4,090 msec user, 190 msec system
; cpu time (total)  23,360 msec user, 750 msec system
; real time  26,117 msec
```

- Consistency checking:

```
; cpu time (non-gc) 63,710 msec (00:01:03.710) user, 3,720 msec system
; cpu time (gc)     11,550 msec user, 420 msec system
; cpu time (total)  75,260 msec (00:01:15.260) user, 4,140 msec system
; real time  101,133 msec (00:01:41.133)
```

- Search for an appropriate convex ordering:

```
; cpu time (non-gc) 430 msec user, 40 msec system
; cpu time (gc)     660 msec user, 60 msec system
; cpu time (total)  1,090 msec user, 100 msec system
; real time  1,774 msec
```

6.3 Some practical examples

6.3.1 A steel structure problem from civil engineering

The introductory example of Figure 1.4 relates to the construction of a floor in a school building, consisting of a concrete slab on steel beams . In this example, four main independent variables, beam depth(H_b), slab thickness(H_s), beam span(W) and beam spacing(S)

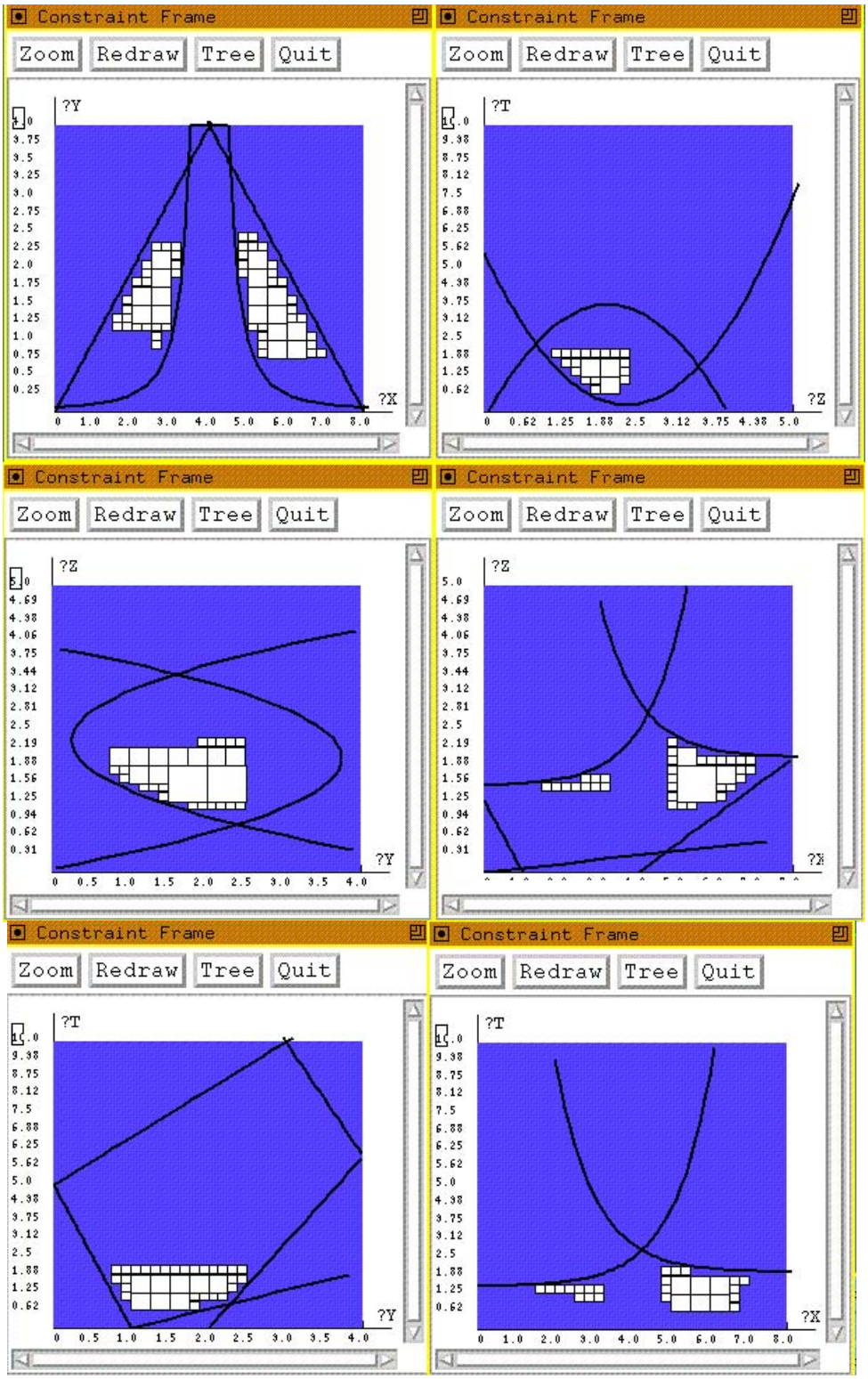


Figure 6.2: *The quadtrees computed by $\mathcal{T}$-PC2 for the binary example*

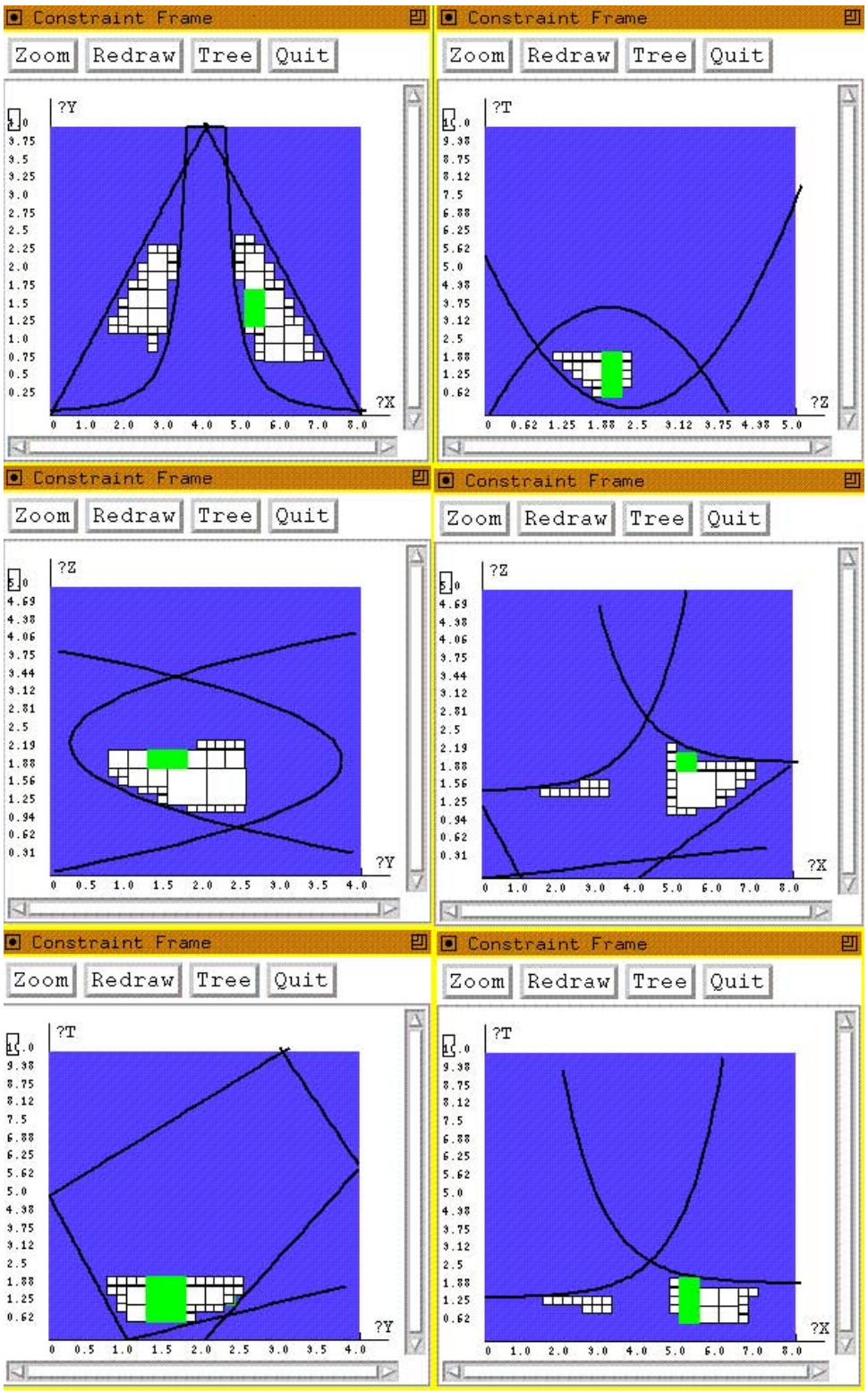


Figure 6.3: (x, y, z, t) *is a* (x)*-convex ordering. Interval solutions can be derived backtrack-free (the green regions correspond to a particular interval solutions derived without backtracking)*

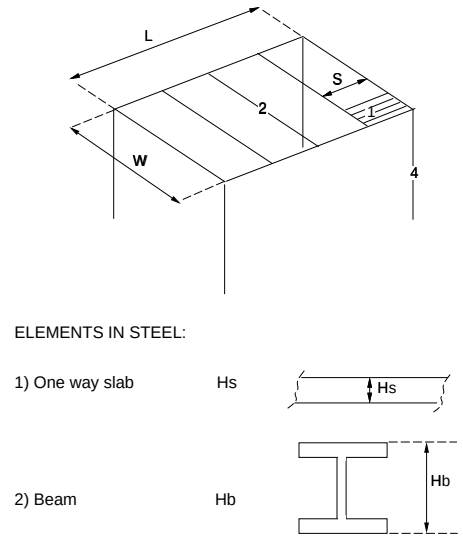


Figure 6.4: *The steel structure problem*

are linked together through the following non-linear constraints:

Structural safety:

$$
\begin{aligned}
H_s &> 137.70 - 8633.10^{-3}S + 5511.10^{-8}S^2 - 835810^{-10}S^3 \\
H_b &> 41.383\left(\frac{1.1 * 2.35 * p * W^2}{10^6 C * f_y}\right)^{0.3976} \\
p &= 1.3\left[\frac{(P_{sw(slab)} + P_{dl}) * S}{1000} + 0.000074\left(\frac{W}{24}\right)^{1.5161}\right]
\end{aligned}
$$

S*erviceability* :

$$
H_b > \left[\frac{P_{llst} * S}{1000}.\frac{D * 350 * W^3}{0.1545 * 384 * E_s}\right]^{0.2839}
$$

P_{llst} is the short term live load which equals to $4kN/m^2$ for schools; and E_s equals to $210000N/mm^2$ for steel. Traditionally, an engineer works through these equations hierarchically; at no time is the complete solution set known. Exploration of possible solutions is done point by point according to the experience of the designer.

Using our method, a large space of potential solution can be made explicit.

We start by transforming the original problem into a ternary equivalent one:

$$
\begin{aligned}
u &< (3.18.10^{-5}H_s + 0.0054)S \\
H_s &> 137.70 - 8633.10^{-3}S + 5511.10^{-8}S^2 - 835810^{-10}S^3 \\
p &= u + 9.62.10^{-5}(0.0417W)^{1.5161}
\end{aligned}
$$

$$H_b > 0.77(p.W^2)^{0.3976}$$

$$H_b > 0.0168(p.W^3)^{0.2839}$$

Figure 6.5 shows the octree representation constructed for each relation. After enforcing (3,2)-relational consistency, the solution space reduces to the regions described by Figure 6.6. The (3,2)-relationally consistent octrees are not convex but (s, w, h_b, p, h_s, u) is identified as a directional (x,y)-convex ordering (see section 5.3.2). The solutions of the problem can therefore be derived backtrack-free, which produces the following instantiations:

```
solution 1:
-----------
?S = [1959.38 , 1981.25],
?W = [5953.12 , 6281.25],
?HB = [267.19 , 275.0],
?P = [18.28 , 19.38],
?HS = [120.31 , 125.0],
?U = [17.81 , 22.5]

solution 2:
-----------
?S = [1981.25 , 2003.12],
?W = [5953.12 , 6281.25],
?HB = [267.19 , 275.0],
?P = [18.28 , 19.38],
?HS = [120.31 , 125.0],
?U = [17.81, 22.5]
.................
```

The prototype implementation on a Silicon Graphics Iris 4D using Allegro Common Lisp 4.2 gives the following statistics:

- Construction of octrees:

```
; cpu time (non-gc) 238,510 msec (00:03:58.510) user, 2,080 msec system
; cpu time (gc)     18,380 msec user, 500 msec system
; cpu time (total)  256,890 msec (00:04:16.890) user, 2,580 msec system
; real time  270,401 msec (00:04:30.401)
```

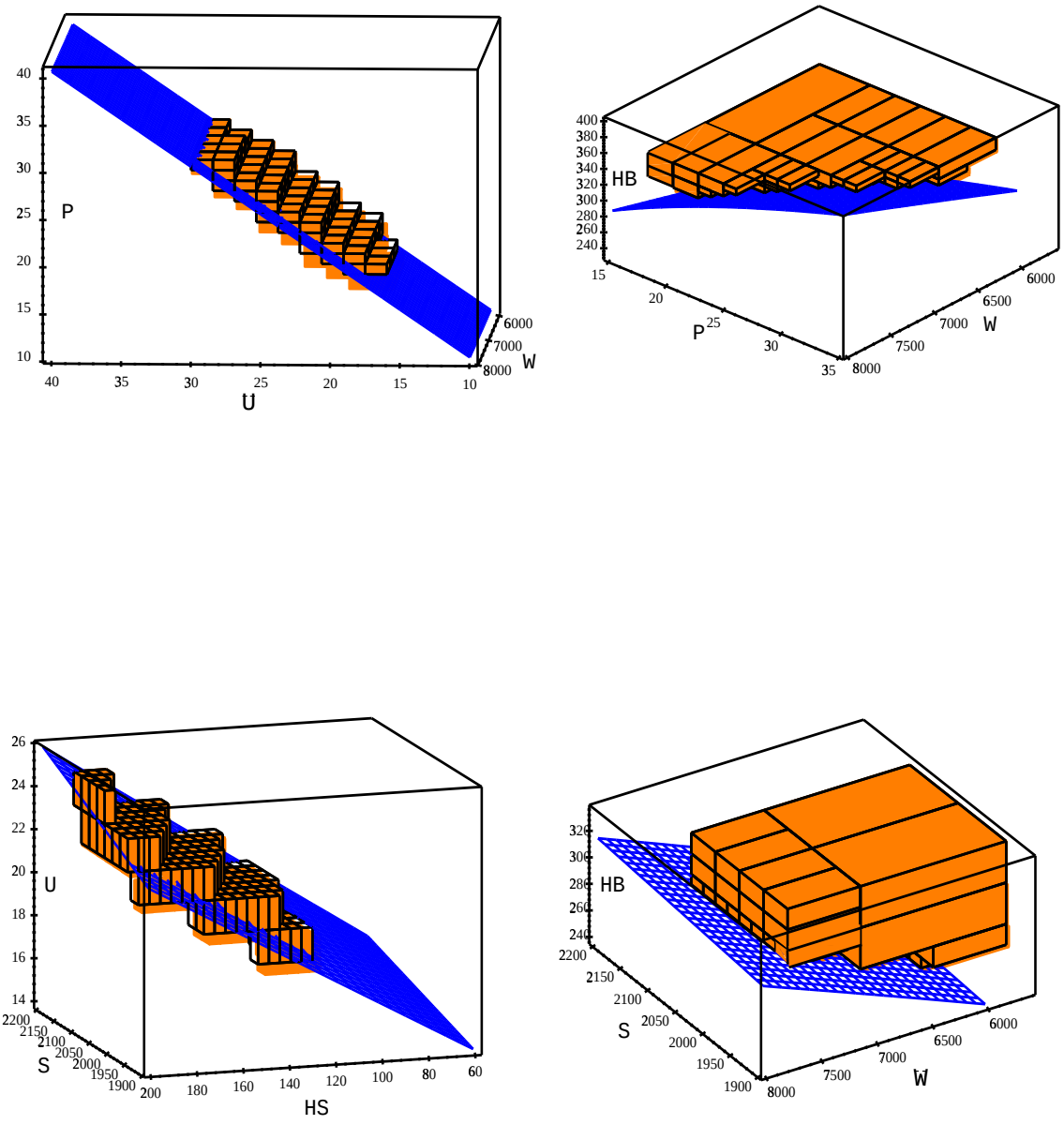


Figure 6.5: *Some octrees constructed for the steel structure problem*

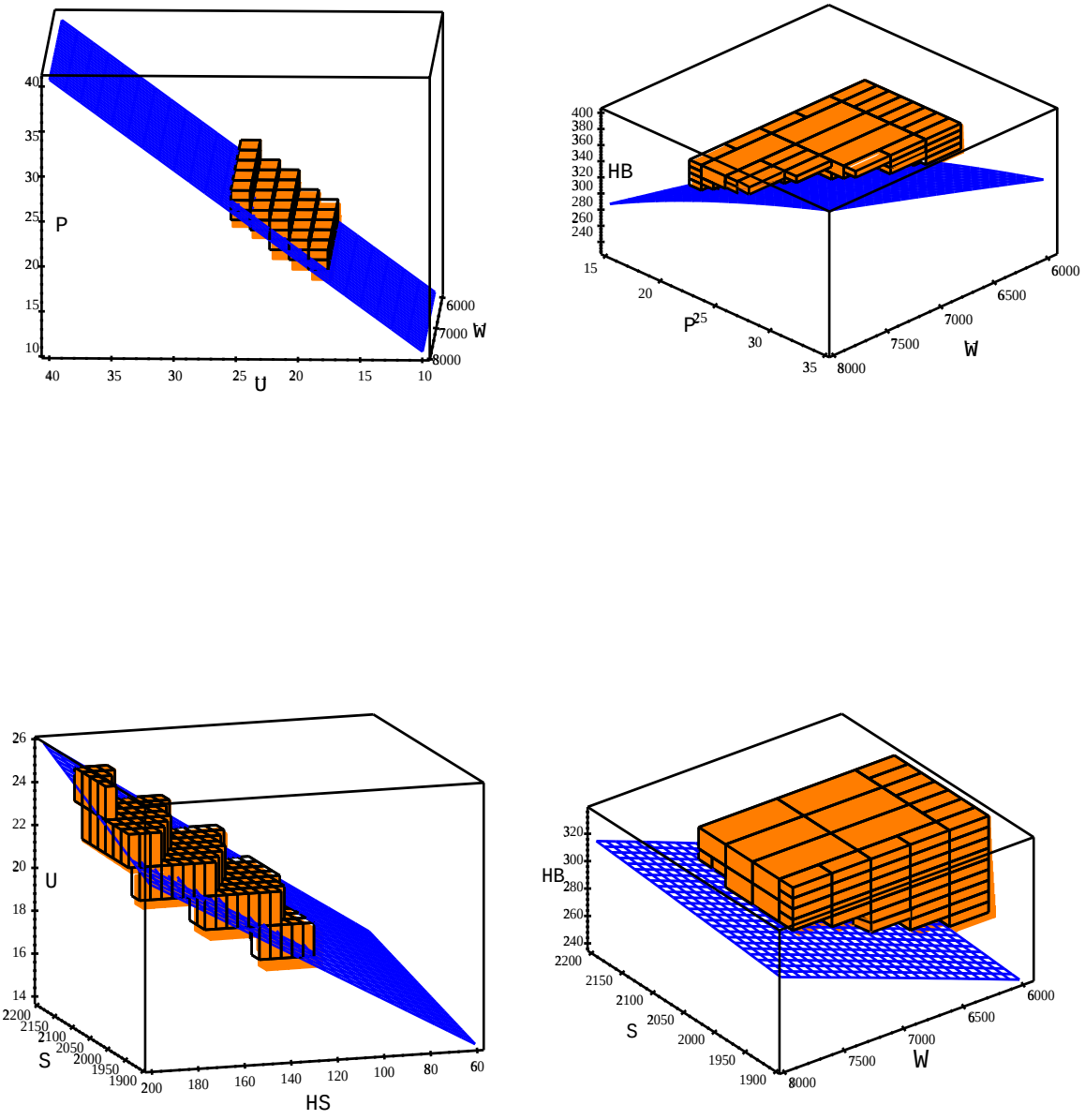


Figure 6.6: *3D projections of the globally consistent solution space*

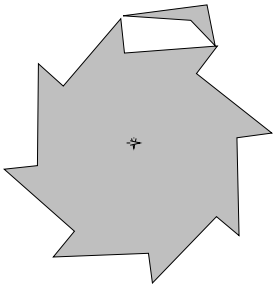


Figure 6.7: *An escapement is a kinematic pair, consisting of a wheel and a pawl, used in mechanical clocks*

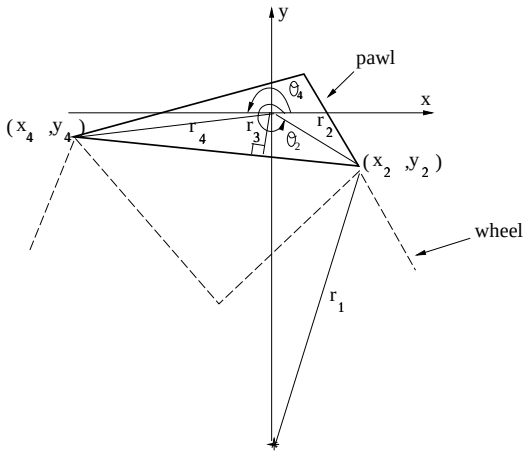


Figure 6.8: *Variables involved when designing an escapement*

- Consistency checking:

```
; cpu time (non-gc) 756,272 msec (00:12:36.270) user, 1,310 msec system
; cpu time (gc)     34,400 msec user, 630 msec system
; cpu time (total)  775,660 msec (00:02:55.660) user, 1,940 msec system
; real time  792,612 msec (00:13:12.612)
; space allocation:
 18,034,122 cons cells,;  162 symbols, 87,435,256 other bytes
```

6.3.2 A mechanical design problem

A kinematic pair is a mechanism consisting of two moving parts, whose function is generated by the interaction of their shape [130]. In this example, we consider the case of

an escapement used in mechanical clocks (see Figure 6.7). Designing a kinematic pair consists of determining the suitable shapes for the components, such that the desired function is achieved. The appropriate shapes are discovered through dimensional modifications obeying the metric constraints. In the case of an escapement, these constraints guarantee that the pawl touches the wheel without blocking its motion, which translates into the following constraints system (see Figure 6.8):

$$
\begin{aligned}
d &< r_1 + r_2 \\
d &> abs(r_1 - r_2) \\
d &> r_1 + r_3 \\
0 &< \frac{r_2(\sin\theta_2 \cos\theta_4 - \cos\theta_2 \sin\theta_4)}{\sqrt{r_2^2 + r_4^2 - 2r_2 r_4(\cos\theta_2 \cos\theta_4 + \sin\theta_2 \sin\theta_4)}} \\
d &< r_1 + r_4 \\
d &> abs(r_1 - r_4) \\
r_3 &= \frac{abs(r_2 r_4 \cos\theta_4 \sin\theta_2 - r_2 r_4 \cos\theta_2 \sin\theta_4)}{\sqrt{(r_2 \cos\theta_2 - r_4 \cos\theta_4)^2 + (r_2 \sin\theta_2 - r_4 \sin\theta_4)}} \\
x_2 &= r_2 \cos\theta_2 \\
x_4 &= r_2 \cos\theta_4 \\
y_2 &= y_4
\end{aligned}
$$

The other teeth of the wheel are derived by rotation and symmetry.

While designing a kinematic pair, it is interesting to identify the solution regions (i.e., the different alternatives of shapes) for which the qualitative behavior of the device does not change. This makes it possible to identify the spectrum of design alternatives exhibiting the desired behavior. In practice however, since the constraints involved are highly non linear, designers only explore pointwise alternatives by selecting single landmark parameters for modification and projecting the constraints onto landmark values of this parameter.

Using our method, several distinct alternatives can be proposed to the designer. Figure 6.9 shows some of the octrees constructed for solving the problem. Figures 6.10 shows the same octrees after the (3,2)-relational consistency has been enforced. The transformation of the original problem into a ternary one required the introduction of three

auxiliary variables (removed by projection from the octrees of Figures 6.10 and 6.9). Here again, the octrees computed by (3,2)-relational consistency are not convex. However $(x_4, \theta_4, r_4, x_2, \theta_2, r_2, r_3, r_1, d, w, w_p, u)$ (where w, w_p and u are the additional variables) is identified as a directionally (x,y)-convex ordering and the solutions can be derived backtrack-free.

6.3.3 An interval method benchmark—illustration of local consistency methods—

Arc consistency techniques based on 2^k-trees can also be used for computing interval approximations of feasibility spaces.

We give in the following an example of interval approximations produced by the algorithm $\mathcal{T}$-AC3 of section 4.3.1. The example is borrowed from [7] and comes from a robotic kinematic problem. It is considered as an interesting benchmark for interval methods [7].

$$\begin{aligned}
s_2 \times c_5 \times s_6 - s_3 \times c_5 \times s_6 - s_4 \times c_5 \times s_6 + c_2 \times c_6 + c_3 \times c_6 + c_4 \times c_6 &= 0.4077 \\
c_1 \times c_2 \times s_5 + c_1 \times c_3 \times s_5 + c_1 \times c_4 \times s_5 + s_1 \times c_5 &= 1.9115 \\
s_2 \times s_5 + s_3 \times s_5 + s_4 \times s_5 &= 1.9791 \\
c_1 \times c_2 + c_1 \times c_3 + c1 \times c_4 + c_1 \times c_2 + c_1 \times c_3 + c_1 \times c_2 &= 4.0616 \\
s_1 \times c_2 + s_1 \times c_3 + s_1 \times c_4 + s_1 \times c_2 + s_1 \times c_3 + s_1 \times c_2 &= 1.7172 \\
s_2 + s_3 + s_4 + s_2 + s_3 + s_2 &= 3.9701 \\
s_i^2 + c_i^2 &= 1 (1 \leq i \leq 6)
\end{aligned}$$

The initial intervals are [-1 , 1]. After nine iterations (˜ 900 seconds using our prototype implementation), $\mathcal{T}$-AC3 reaches quiescence and produces the following interval approximations:

```
c1= [0.9171753 , 0.9278564]
s1= [0.3828125 , 0.4023438]
c2= [0.8730469 , 0.8876953]
s2= [0.4818115 , 0.5048828]
c3= [0.671875  , 0.7128906]
s3= [0.7252274 , 0.7710228]
c4= [0.2512207 , 0.33496094]
s4= [0.9327087 , 0.9663544]
c5= [0.2930908 , 0.3297272]
s5= [0.9499112 , 0.959929]
```

```
c5= [0.2930908 , 0.3297272]
s5= [0.9499112 , 0.959929]
c6= [0.0 , 0.1875]
s6= [0.75 , 1]
```

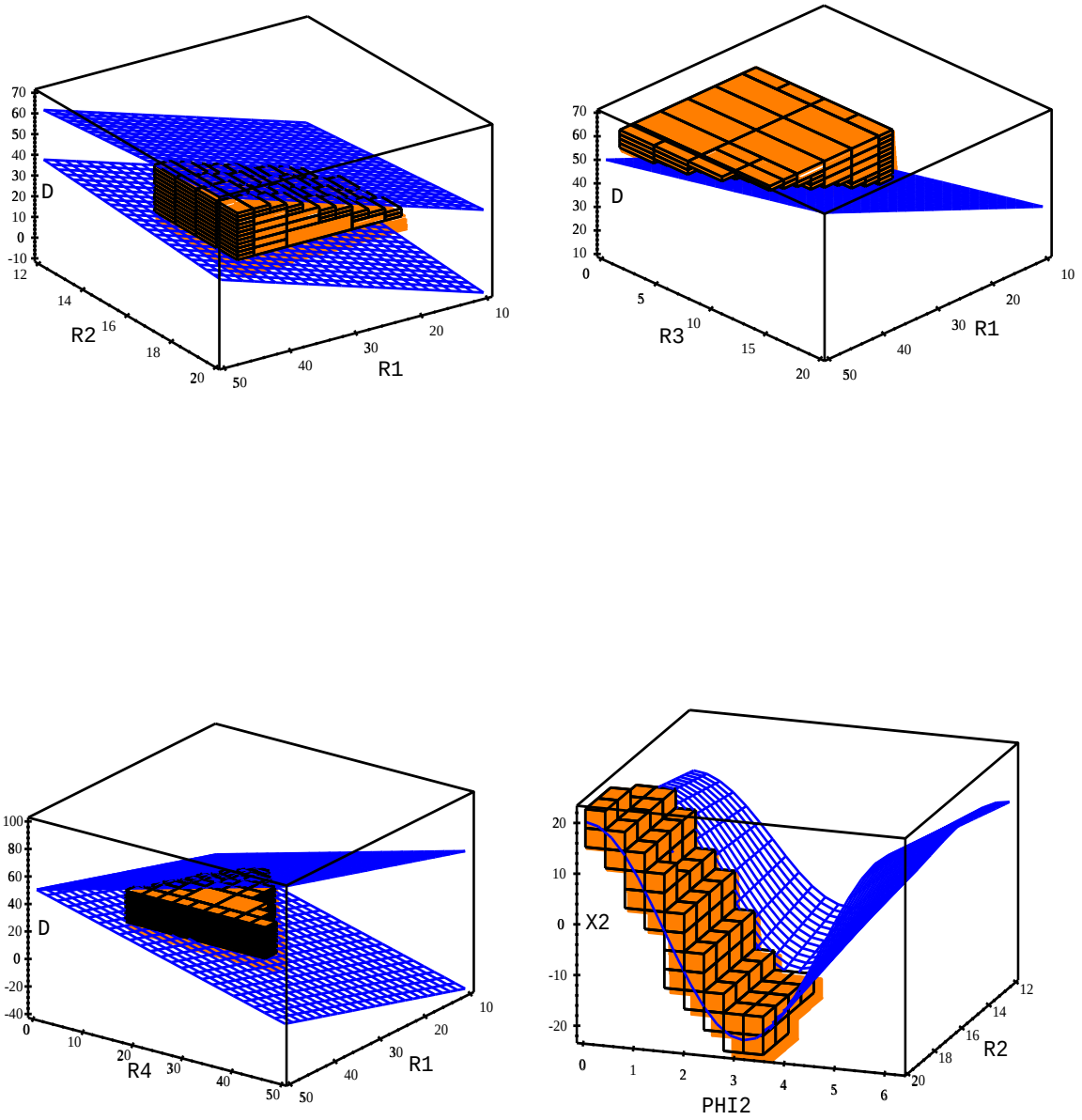


Figure 6.9: *Some octrees constructed for the mechanical design problem*

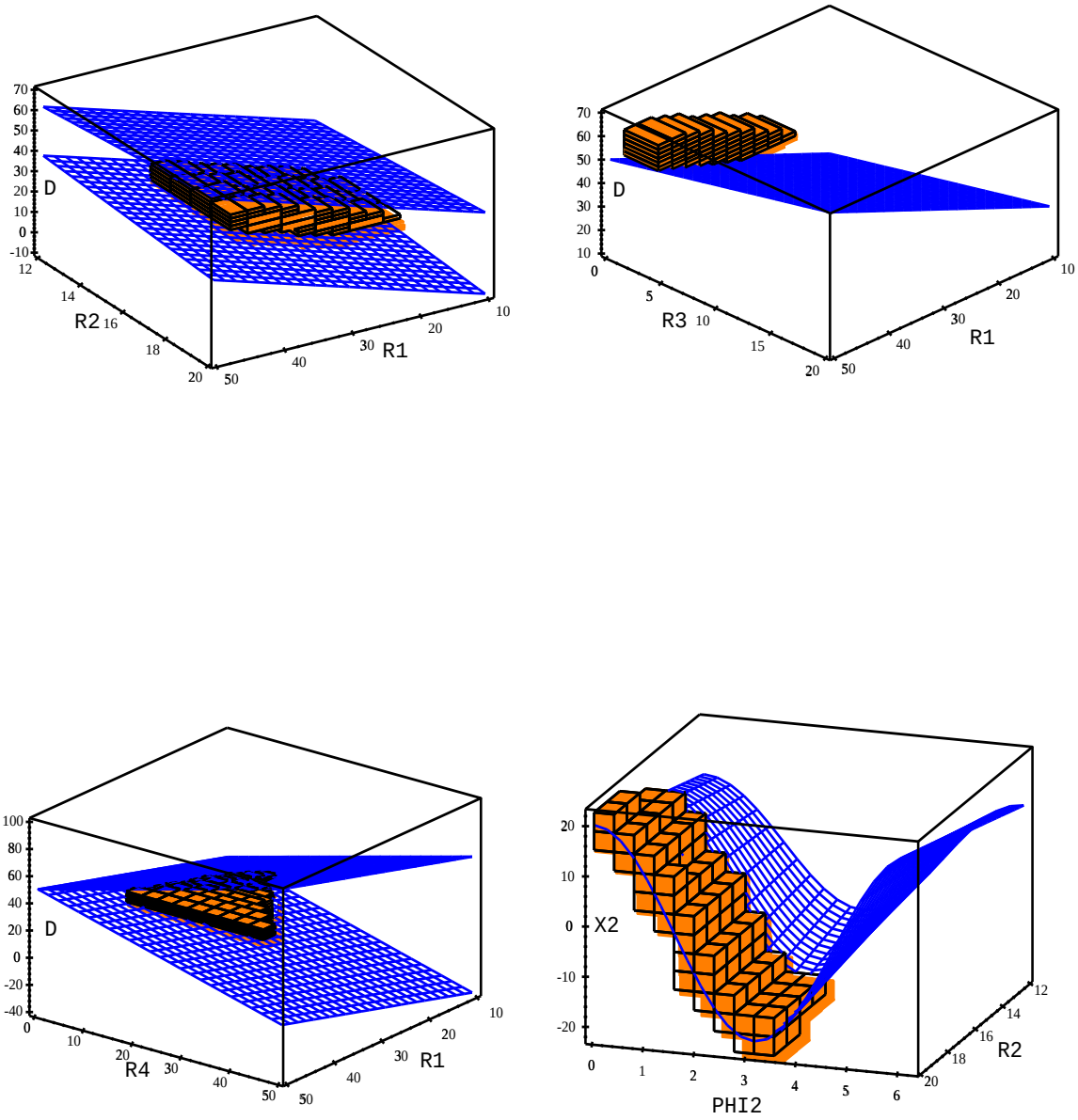


Figure 6.10: *3D projections of the globally consistent solution space computed for the mechanical design problem*

Optimized implementations of the state of the art interval methods takes respectively 2395 seconds [61], and 454 seconds [7] to solve the problem. This is rather encouraging for the 2^k-tree based algorithms which are not specifically dedicated to perform interval narrowing on systems of *equations* . Using $\mathcal{T}$-AC3 the variable dependency problem may prevent the sharp narrowing of some variables (for example c_6 and s_6 in our example), however since the 2^k-tree based algorithms for arc-consistency offer more guarantees in terms of robustness and convergence they can be used to locate safe starting regions for locally convergent methods.

6.4 Summary

In this chapter we illustrated how we make use of our techniques to solve constraint satisfaction problems with continuous variables. Practical experiences on real-world design examples were reported.

Chapter 7

Conclusion

In this chapter we draw the main conclusions of this research. The principal contributions are reviewed, and directions for further work are suggested.

7.1 Summary of major results

Many practical problems need to reason about ranges of feasible values rather than on pointwise solutions. Traditional interval-based approximation techniques use a purely numerical handling of constraints and are thus inevitably confronted by convergence and stability problems. This explains why the problem reduction techniques based on local consistency checking have been the subject of little research, and that major contributions to the field have been restricted to the design of algorithms for low degrees of consistency.

This dissertation defends the thesis that the limited performance of interval-based consistency techniques is only partly due to the analytical complexity inherent to solving numerical systems and shows that:

- an appropriate representation of numerical constraints enables the design of a reliable algorithm for a higher degree of consistency than arc consistency
- general conditions can be identified under which global consistency is tractable for continuous CSPs

Our main contributions can be summarized as follows:

- we propose a partially discretized representation of continuous constraints based on a hierarchical decomposition of feasibility spaces into 2^k-trees. This representation,

commonly used in computer vision and image processing, conveys an easy implementation of reliable constraint combination operators. These operators are based on a logical and explicit rather than numerical and implicit handling of solution regions

- we give simple and stable algorithms for computing labelings of arbitrary degrees of consistency using the 2^k-tree representation of constraints
- we exhibit interesting convexity properties under which global consistency is tractable and we propose simple algorithms for verifying these properties on 2^k-trees. These results, first stated for binary problems, extend previous works on discrete and continuous temporal CSPs
- we introduce the concept of $(r, r-1)$-relational consistency which enables a tractable extension of the result on binary problems to the case of n-ary constraints. This leads to polynomial-time algorithms for computing globally consistent labelings for a large class of constraint satisfaction problems with continuous variables
- we propose a general classification of continuous CSPs based on topological characteristics of their solution spaces
- we devise search and decomposition strategies for reducing the complexity of solving problems verifying no interesting convexity properties

A more detailed discussion of each contribution is given at the end of each chapter.

7.2 Applicability and limitations

The framework we define for handling numerical constraints is meant to target the problems where having a complete and sound description of the set of alternative solutions is desirable, within a limited degree of precision.

We have demonstrated using practical design problems, that the constraint representation along with the consistency algorithms we propose respond satisfactorily to this objective.

In terms of practicality, the major advantages of the consistency methods described in this dissertation lie in their generality and stability as:

- constraints of arbitrary types or arities can be handled reliably
- cyclic constraint networks do not pose convergence problems
- the need for numerical tools is restricted to simple subproblems
- the precision of results can be tuned deterministically
- complex pre-conditioning steps are not required

These features make our method attractive for a large variety of engineering problems where the numerical solution of constraint satisfaction problems condition complex decision processes.

However, our techniques are naturally more time and space demanding than the traditional interval-based local consistency methods, which have the purpose of constructing rapidly enclosing approximation with no particular concern about global soundness. A more detailed comparison with related works is given in section 4.5.

The engineering orientation of our objectives has led us to restrict the applicability conditions as follows:

- a limited degree of precision must exist under which the results have no significance. This means that our methods are cast as approximation tools rather than as exact mathematical solvers
- the domain of variables must be bounded,

Other limitations are inherent in our technological options:

- n-ary constraints cannot be treated as a whole but must be decomposed into ternary ones. The question of optimizing this decomposition by minimizing the number of additional variables requires further investigation
- the methods developed are adapted to solving *sets* of simultaneous constraints. Their application require tools for finding the zeros of individual equations, a task for which they are consequently not adapted
- discretization may not preserve the topological properties of the solution space when identifying the feasible regions requires reaching the limits of resolution

7.3 Further research

7.3.1 Parallel Implementation

Both of the 2^k-trees construction and convexity checking tasks discussed in chapters 3 and 5 respectively are well suited to parallel implementation using architectures with multiple processors. In quadtrees (respectively octrees) construction, the tasks of determining the color of nodes in two separate quadrants (respectively octants) of the same level of decomposition are completely independent and can therefore be carried out in parallel.

Similarly the tasks of checking for the (x)-convexity (respectively (x,y)-convexity) of two different slices in a quadtree (respectively an octree) can be executed independently.

Recent studies in computer vision also propose the parallel implementations of certain operations on 2^k-trees (like the composition operation).

7.3.2 Search in numerical spaces

In section 5.5.2 we propose a backtracking search schema for solving multi-convex CCSPs. This schema opens up the possibility of elaborating and evaluating different search heuristics. It also offers the opportunity of studying the suitability in continuous domains of the numerous search strategies developed for discrete problems.

7.3.3 Classification of non-convex problems

A broad classification of continuous CCSPs, in terms of computational complexity is proposed is section 5.5, where convex CCSPs are recognized as easy instances. In discrete domains, recent studies (see [20]) have tried to identify order parameters characterizing the complexity of general constraint satisfaction problems. It would be interesting to know whether similar approaches could give more insight into the complexity of multi-convex and non-convex CCSPs, and could aid in distinguishing the easy instances from the hard ones in this large class of problems.

7.3.4 Decomposition of non-convex problems

A heuristic method is proposed in section 5.5.4 for decomposing non convex problems into multi-convex ones. This method simply tries to gather in a partially convex subset the

maximum number of 2^k-tree nodes such that the convexity property is verified in a given direction. For each partially convex isolated subregion, the same process is iterated in other directions until the convexity property is enforced in every possible direction. This method was devised for the purpose of only showing the feasibility of such a decomposition and does not have any claim in terms of optimality and efficiency. Investigating more elaborate decomposition strategies could be an interesting direction for further prospection. Computational geometry [110] for example, proposes several decomposition methods whose goal is to partition a given geometric object into collection of simpler primitive geometric objects (such as triangles, convex polygons, star-polygons, etc...). Some methods in particular, are devoted to constructing decompositions with the minimum numbers of parts [79]. It would be interesting to investigate whether analog methods could be designed for decomposing the geometric objects determined by quadtrees and octrees.

7.3.5 Result interpretation

When a concise description of the whole solution set is computed, one issue that presents itself is how this description can be expressively presented to the user. Our prototype implementation displays the set of possible solutions either by projecting the solution space in 2D or 3D, or by listing the possible feasible interval instantiations for the variables. The interpretation of multi-dimensional results concerns itself more with human-computer interaction. Further interfacing investigations are required to determine expressive representations allowing the user to explore the solution space and capture the nature of possible choices as easily as possible.

7.3.6 Other optimization issues

Several practical improvements are possible concerning the implementation issues:

- the partial consistency techniques we propose in chapter 4 are implicitly based on a binary search. They are consequently stable and robust but might be slow to produce sharp bounds of the feasibility domain. Other interval techniques based on locally convergent methods could refine the results and complement our algorithms once they have properly located the effective solution regions

- equalities are by nature more space demanding than inequalities since they preclude condensation of nodes in 2^k-trees. For certain types of problems a possible improvement would be to use advanced algebraic manipulation, available in commercial packages such as Macsyma [43] or Mathematica [94], to remove as many equalities as possible before starting the resolution process

- the (3,2)-relational consistency algorithm presented in chapter 6 is a brute force implementation of the new consistency concept we propose. This algorithm can be improved by computing only the binary extensions affected by a revision step rather that the whole set of possible binary extensions

- the decomposition of n-ary constraints into ternary ones pose the problem of reducing as much as possible the number of additional variables. Further investigation is required for automating this task. Another issue of interest is, whether using appropriate interval methods for solving single multivariate equations one can automatically construct 2^k-trees of higher arity than octrees. This would enable the decomposition of n-ary constraints into constraints of higher arity than three, thus limiting the number of additional auxiliary variables and reducing the complexity of local consistency techniques

- the 2^k-tree decomposition of a continuous relation imposes a uniform precision on all the variables it involves. It would be interesting to study whether alternative structures (such as k^d-trees), enabling a distinct precision for each individual variable, lend themselves to an easy implementation of the logical constraint combination operators required by consistency techniques

7.4 Conclusions

A myriad of methods, ranging from mathematical and operations research to interval arithmetic and local consistency methods, are devoted to solving numerical systems. Due to the difficulty of dealing with arbitrary numerical constraints, it is not surprising that none of the techniques surveyed in this thesis nor the technique we propose, provides satisfactory solutions in all cases. Some methods are designed to respond efficiently to the needs of particular problems. Others, more general even if less performant, further our

understanding of numerical reasoning by widening the field of potential applicability. We believe that a synergetical use of these concepts along with appropriate search strategies and problem modeling tools are the key to the emergence of powerful solvers. This thesis extends the contribution of consistency techniques to this issue.

Bibliography

[1] Alefeld G. On the convergence of some interval-arithmetic modifications of Newton's methods. In *SIAM Journal of numerical analysis 21(2)*. 1984.

[2] Alefeld G. and Herzberger J. *Introduction to interval computations*. Academic Press, 1983.

[3] Allgower E. and Georg K. Simplicial and continuation methods for approximating fixed points and solutions to systems of equations. In *SIAM review 22*. 1980.

[4] Avriel M. *Non linear programming: analysis and methods*. Prentice-Hall, 1976.

[5] E. Tyugu B. Mayoh and J. Penjam, editors. *Constraint Programming*. Springer-Verlag (NATO ASI Series), 1994.

[6] Bellman R. *Adaptive control processes: a guided tour*. Princeton University Press, 1961.

[7] Benhamou F., McAllester D., and van Hentenryck P. CLP(intervals) revisited. In *Proceedings of the International Logic Programming Symposium 94*, 1994.

[8] Benhamou F. and Older W. Applying interval arithmetic to real integer and boolean constraints. In *Journal of Logic Programming*. 1994.

[9] Bessiere C. and Cordier M-O. Arc-consistency and arc-consistency again. In *Proceedings of the 11th National Conference on AI*, 1993.

[10] Bliek C. *Computer methods for design automation*. PhD thesis, Department of ocean engineering, MIT, 1992.

[11] Bowen J. and Bahler D. Conditional existence of variables in generalized constraint networks. In *Proceedings of the 9th National Conference on AI*, 1991.

[12] Brooke A., Kendrick D., and Meerans A. *GAMS 2.25 — A User's Guide.* The Scientific Press, 1992.

[13] Broyden C.G. A class of methods for solving non-linear simultaneous equations. In *Math. Comp. 19.* 1978.

[14] B. Buchberger. Gröebner bases: an algorithmic method in polynomial ideal theory. In N.K. Bose, editor, *Recent trends in multidimensional systems theory.* D. Riedel Publ. Comp., 1983.

[15] Buchberger B. *An algorithm for finding a basis for the residue class ring of a zero-dimensional polynomial ideal.* PhD thesis, Institut fur mathematik, Universitat Insbruck, 1965.

[16] Buchberger B., Collins G.E., and Loos R. *Computer algebra: symbolic and algebraic computation, 2nd edition.* Springer-Verlag, 1983.

[17] Char B.W., Geddes K.O., Gonnet G.H., Leong B.L., Monagan M.B., and Watt S.M. *Maple V reference manual.* Springer-Verlag, 1993.

[18] Cheesman P., Kanefsky B., and Taylor W.M. Where the really hard problems are. In *Proceedings of the 12th International Joint Conference on AI*, 1991.

[19] Chiu C.K. and Lee T.W. Towards practical interval constraint solving in logic programming. In *Proceedings of the 11th International Logic Programming Symposium*, 1994.

[20] Choueiry B.Y. *Abstraction methods for resource allocation.* PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 1994.

[21] Cleary J.G. Logical arithmetic. In *Future Computing Systems, 2(2).* 1987.

[22] Clowes M.B. On seeing things. In *Artificial Intelligence, 2.* 1971.

[23] Collins G. E. and Hong H. Partial cylindrical decomposition for quantifier elimination. In *Journal of Symbolic Computation, 12(3).* 1991.

[24] Collins G.E. Quantifier elimination for the elementary theory of real closed fields by cylindrical algebraic decomposition. In *Lectures notes in computer sciences 33.* Springer Verlag, 1975.

[25] Colmerauer A. An introduction to PrologIII. In *Communications of the ACM, 33(7).* 1990.

[26] Colmerauer A. Naive solving of non linear constraints. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint logic programming — Selected research.* The MIT press, 1993.

[27] Cooper M.C. An optimal k-consistency algorithm. In *Artificial Intelligence, 41,* 1989.

[28] Cooper M.C., Cohen D.A., and Jeavons P.G. Characterizing tractable constraints. In *Artificial Intelligence, 65.* 1994.

[29] Czapor S. R. Solving algebraic equations: combining Buchberger's algorithm with multivariate factorization. In *Journal of Symbolic Computation, 7(1).* 1989.

[30] Dague P. Numeric reasoning with relative order of magnitude. In *Proceedings of the 11th National Conference on AI,* 1993.

[31] Danzig G. B. *Linear programming and extensions.* Princeton University Press, 1965.

[32] Davenport J.H. Looking at a set of equations. In *Bath computer science technical report 87-06.* School of mathematics, University of Bath, England, 1987.

[33] David P. When functional and bijective constraints make a CSP polynomial. In *Proceedings of the 13th International Joint Conference on AI,* 1993.

[34] Davis E. Constraint propagation with interval labels. In *Artificial Intelligence 32,* 1987.

[35] Dechter R. From local to global consistency. In *Proceedings of the 8th Canadian Conference on AI,* 1990.

[36] Dechter R. From local to global consistency. In *Artificial Intelligence 55,* 1992.

[37] Dechter R. and Pearl J. Network-based heuristics for constraint satisfaction problems. In *Artificial Intelligence, 34*. 1988.

[38] Dechter R., Meiri i., and Pearl J. Temporal constraint networks. In *Artificial Intelligence 49(1-3)*, 1990.

[39] Dechter R. and Pearl J. Directed constraint networks: a relational framework for causal modeling. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, 1991.

[40] Dennis J.E. Jr. A brief introduction to quasi-Newton methods. In G.H. Golub and J. Oliger, editors, *Numerical analysis*. 1978.

[41] Deville Y. and van Hentenryck P. An efficient arc consistency algorithm for a class of CSP problems. In *Proceedings of the 12th International Joint Conference on AI*, 1991.

[42] Dincbas M., van Hentenrick P., Aggoun A. Simonis, Graf T., and Berthier F. The constraint logic programming language CHIP. In *Proceedings of the International conference on fifth generation computer systems*, Tokyo, Japan, 1988.

[43] Drinkard R.D. and Sulinski N.K. *MACSYMA: A program for computer Algebraic Manipulation (demonstration and analysis)*. Naval Underwater Systems Center, Newport, RI, 1981.

[44] Faltings B. Arc consistency for continuous variables. In *Artificial Intelligence 65(2)*, 1994.

[45] Faltings B., Haroud Sam D., and Smith I. Dynamic constraint propagation with continuous variables. In *Proceedings of the 10th European Conference on AI*, 1992.

[46] Freuder E.C. Synthesizing constraint expressions. In *Communications of the ACM, 21*, 1978.

[47] Freuder E.C. A sufficient condition for backtrack-free search. In *Journal of the ACM, 29*, 1982.

[48] Freuder E.C. A sufficient condition for backtrack-bounded search. In *Journal of the ACM, 32*, 1985.

[49] Freuder E.C. and Hubbe P.D. Extracting constraint satisfaction subproblems. In *ECAI'94 Workshop on constraint satisfaction issues raised by practical applications W19*, 1994.

[50] Fröberg C.E. *Numerical mathematics — Theory and computer applications.* The Benjamin/Cummings publishing company Inc., 1985.

[51] Fuji Y., Ichida K., and Ozasa M. Maximization of multivariate functions using interval analysis. In *Interval mathematics 1985: proceedings of the international symposium.* Springer-Verlag, 1985.

[52] Gargantini I. An effective way to represent quadtrees. In *Communications of the ACM, 25(12)*, 1982.

[53] Gill P.E. and Murray W. *Numerical methods for constrained optimization.* Academic Press, 1974.

[54] Giovini A., Mora T., Niesi G., Robbiano L., and Traverso C. "one sugar cube, please" or selection strategies in the Buchberger algorithm. In S.M. Watt, editor, *Proceedings of ISSAC'91.* 1991.

[55] Grigor'ev D.Y. The complexity of deciding Tarski algebra. In *Journal of Symbolic Computations, 5(1,2).* 1988.

[56] Grigor'ev D.Y. and Vorobjov N.N. Solving systems of polynomial inequalities in subexponential time. In *Journal of Symbolic Computations, 5(1,2).* 1988.

[57] Guesgen H.W. and Hertzberg J. A perspective of constraint-based reasoning. In *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 1992.

[58] Hager G.D. Task-directed computation of qualitative decisions from sensor data. In *IEEE Transactions on Robotics and Automation, 10(4)*, 1994.

[59] Han C. and Lee C. Comments on Mohr and Henderson's path consistency algorithm. In *Artificial Intelligence 36*, 1988.

[60] Hansen E. and Greenberg R.I. An interval-Newton method. In *Appl. Math. Comput. 12.* 1983.

[61] Hansen E. and Sengupta S. Bounding solutions of systems of equations. In *BIT, 21.* 1981.

[62] Hansen E. and Smith R. Interval arithmetic in matrix computations. In *SIAM Journal of numerical analysis 4:1-9.* 1967.

[63] Haralick R.M. and Shapiro L.G. The consistent labeling problem: Part I. In *IEEE Transactions PAMI, 1(2).* 1979.

[64] Haralick R.M. and Shapiro L.G. The consistent labeling problem: Part II. In *IEEE Transactions PAMI, 2(3).* 1980.

[65] Haroud Sam D., Boulanger S., Gelle E., and Smith I. Management of conflict for preliminary engineering design tasks. In *AIEDAM, 9(4)*, 1995.

[66] Hearn A.C. *Reduce: User's manual.* RAND Publication, Santa Monica, CA, 1991.

[67] Hellwich M., Kästner H., and Küstner H. *The VNR concise encyclopedia of mathematics, 2nd edition.* van Nostrand Reinhold Company (VNR), 1989.

[68] Hickey T. CLP(F) and constrained ODE's. In *Proceedings of 1994 Workshop on Constraints and Modeling*, 1994.

[69] Hollman J. and Langemyr L. Algorithms for non linear algebraic constraints. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint logic programming — Selected research.* The MIT press, 1993.

[70] Hong H. *Improvements in CAD-based quantifier elimination.* PhD thesis, The Ohio State University, Columbus, 1990.

[71] Hong H. Non-linear real constraints in constraint logic programming. In *Springer lecture notes in computer science 632: International conference on algebraic and logic programming.* Springer Verlag, 1992.

[72] Hong H. RISC-CLP(Real): logic programming with non-linear constraints over the reals. In *Constraint logic programming - selected research.* The MIT press, 1993.

[73] Huffman D.A. Impossible objects as nonsense sentences. In R. Metzer and D. Michie, editors, *Machine Intelligence, 6.* Elsevier, 1971.

[74] Hunter G.M. Operations on images using quad trees. In *IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-1(2)*, 1979.

[75] Hyvönen E. Constraint reasoning based on interval arithmetic: the tolerance propagation approach. In *Artificial Intelligence 58(1-3)*, 1992.

[76] Jaffar J., Michaylov S., Stuckey P., and Yap R. The CLP(R) language and system. In *Technical report RC 16292 (#72336), IBM Research division.* 1990.

[77] Dennis J.E. and Schnabel R.B. *Numerical methods for unconstrained optimization and non linear equations.* Prentice-Hall series in computational mathematics, 1983.

[78] Kearfott R.B. Some tests of generalized bisection. In *ACM Transactions on Mathematical Software, 13(3).* 1987.

[79] Keil J.M. and Sack J.R. Minimum decomposition of polygonal objects. In G.T. Toussaint, editor, *Computational Geometry.* 1985.

[80] Kramer G.A. *Solving geometric constraint systems.* The MIT Press, 1992.

[81] Krawczyk R. Newton-algorithmen zur bestimmung von nullstellen mit fehlerschranken. In *Computing, 4.* 1969.

[82] Krawczyk R. A class of interval-Newton operators. In *Computing, 37.* 1986.

[83] Kumar V. Algorithms for constraint satisfaction problems: a survey. In *AI Magazine, 13(1).* 1992.

[84] Langemyr L. Cylindrical algebraic decomposition and multiple algebraic extensions. In *Proceedings of the 4th IMA conference on the Mathematics of surfaces.* 1990.

[85] Lazard D. Improvement of the projection set for the cylindrical decomposition. In *Proceedings of the Shreeram Abhyakars's 60^{t}h birthday conference on algebraic geometry and applications*, 1990.

[86] Lee J.H.M. and van Emden M.H. Adapting CLP(R) to floating-point arithmetic. In *Proceedings of the 1992 International Conference on Fifth Generation Computer Systems*, 1992.

[87] Lee J.H.M. and van Emden M.H. Interval computation as deduction in CHIP. In *Journal of Logic Programming, 16(3-4)*. 1993.

[88] Lhomme O. Consistency techniques for numeric CSPs. In *Proceedings of the 13th International Joint Conference on AI*, 1993.

[89] Luenberger D.G. *Linear and non-linear programming.* Addison-Wesley Publishing company, 1984.

[90] Mackworth A.K. Consistency in networks of relations. In *Artificial Intelligence, 8*, 1977.

[91] Mackworth A.K. The logic of constraint satisfaction. In *Artificial Intelligence, 58(1-3)*. 1992.

[92] Mackworth A.K. and Freuder E.C. The complexity of some polynomial consistency algorithms for constraint satisfaction problems. In *Artificial Intelligence, 25*. 1985.

[93] Mackworth A.K., Mulder J.A., and Havens W.S. Hierarchical arc consistency: exploiting structured domains in constraint satisfaction problems. In *Computational Intelligence 1*, 1985.

[94] Maeder R. *Programming in Mathematica.* Addison-Wesley Publishing Company Inc., 1990.

[95] Melenk H., Möller H. M., and Neun W. Groebner - A package for calculating Groebner bases. In *Reference manual of the computer algebra system Reduce.*

[96] Meseguer P. Constraint satisfaction problem: an overview. In *AI Communications, 2(1)*. 1989.

[97] Mittal S. and Falkenhainer B. Dynamic constraint satisfaction problems. In *Proceedings of the 8th National Conference on AI*, 1990.

[98] Mohr R. and Henderson T.C. Arc and path consistency revisited. In *Artificial Intelligence 28*, 1986.

[99] Montanari U. Networks of constraints: fundamental properties and applications to picture processing. In *Inform. Scie. 7*, 1974.

[100] Moore R.E. *Interval analysis.* Prentice-Hall, Englewood Cliffs,NJ, 1966.

[101] Moore R.E. *Methods and Applications of Interval Analysis.* SIAM Studies in Applied Mathematics, 1979.

[102] Moore R.E. *Reliability in computing: the role of interval methods in scientific computing.* Academic Press, 1988.

[103] Moore R.E. and Jones S.T. Safe starting regions for iterative methods. In *SIAM Journal of Numerical Analysis, 14(6).* 1977.

[104] Mortenson M. E. *Geometric modeling.* John Wiley & Sons, 1985.

[105] Murray W. *Numerical methods for unconstrained optimization.* Academic press, London, 1972.

[106] Neumaier A. *Interval methods for systems of equations.* Cambridge University Press, 1990.

[107] Older W. and Vellino A. Constraint arithmetic on real intervals. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint logic programming — Selected research.* The MIT press, 1993.

[108] Ortega J.M. and Rheinboldt W.C. *Iterative solution of non-linear equations in several variables.* Academic Press, NY, 1970.

[109] Pesant G. and Boyer M. Adding the power of quadratic constraints. In Alan Borning, editor, *Lecture notes in computer Science 874: Principles and Practice of constraint programming.* Springer Verlag, 1994.

[110] Preparata F.P. and Shamos M.I. *Computational geometry — An introduction.* Springer-Verlag, 1985.

[111] Ratschek H. and Rokne J. *Computer methods for the ranges of functions.* Ellis Horwood Limited, Chichester, 1984.

[112] Ratschek H. and Rokne J. *New computer methods for global optimization.* Ellis Horwood Limited, Chichester, 1988.

[113] Renegar J. On the computational complexity of the first-order theory of the reals. Technical Report 853,854,856, Cornell University, Ithaca, New York, 1989.

[114] Ris F.N. *Interval analysis and applications to linear algebra.* PhD thesis, Oxford university,UK, 1972.

[115] Rossi F., Petrie C., and Dhar V. On the equivalence of constraint satisfaction problems. In *Proceedings of the 9th European Conference on AI*, 1990.

[116] Sacks E. Hierarchical reasoning about inequalities. In *Proceedings of the 7th National Conference on AI*, 1987.

[117] Sakai K. and Aiba A. . CAL: a theoretical background of CLP and its applications. In *Journal of symbolic computations, 8*, 1990.

[118] Samet H. *The design and analysis of spatial data structures.* Addison-Wesley Publishing Company, 1990.

[119] Samet H. *Applications of spatial data structures — Computer graphics, image processing and GIS.* Addison-Wesley Publishing Company, 1993.

[120] Samet H. and Webber R.E. Hierarchical data structures and algorithms for computer graphics, PartI. In *IEEE Computer & graphics (May).* 1988.

[121] Samet H. and Webber R.E. Hierarchical data structures and algorithms for computer graphics, PartII. In *IEEE Computer & graphics (July).* 1988.

[122] Sannella M. The SkyeBlue constraint solver and its applications. In Saraswat and van Hentenryck, editors, *Proceedings of the 1993 Workshop on Principles and Practice of Constraint Programming.* The MIT Press, 1994.

[123] Sato S. and Aiba A. An application of CAL to robotics. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint logic programming — Selected research.* The MIT press, 1993.

[124] Schrag B. The quantity lattice for engineering design. In *Proceedings of the AID'92 Wokshop on Analysis*, Pittsburgh, 1992.

[125] Schrijver A. *Theory of linear and integer programming.* Wiley-Interscience series in discrete mathematics, 1986.

[126] Shaffer C.A. and Samet H. Optimal quadtree construction algorithms. In *Computer Vision, Graphics and Image Processing, 37*, 1985.

[127] Sidebottom G. and Havens W.S. Hierarchical arc-consistency for disjoint real intervals in constraint logic programming. In *Computational Intelligence, 8(4)*, 1992.

[128] Simmons R. "Commonsense" arithmetic reasoning. In *Proceedings of the 6th National Conference on AI*, 1986.

[129] Stoer J. and Bulirsch R. *Introduction to numerical analysis.* Springer Verlag, 1993.

[130] Sun K. *Computer-aided creative mechanism design.* PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 1994.

[131] Sutherland I.E. Sketchpad: A man-machine graphical communication system. In *Proceedings of the IFIPS Spring Joint Conference*, 1963.

[132] Tanimoto S. A constraint decomposition method for spatio-temporal configurations problems. In *Proceedings of the 11th National Conference on AI*, 1993.

[133] Tarski A. *A decision method for elementary algebra and geometry, 2nd edition.* University of California Press, Berkeley, CA, 1951.

[134] Thornton A.C. *Constraint specification and satisfaction in embodiment design.* PhD thesis, Department of Engineering, University of Cambridge, UK, 1993.

[135] Tsang E. *Foundations of constraint satisfaction.* Academic Press, 1992.

[136] van Beek P. Approximation algorithms for temporal reasoning. In *Proceedings of the 11th International Joint Conference on AI*, 1989.

[137] van Beek P. On the minimality and decomposability of constraint networks. In *Proceedings of the 10th National Conference on AI*, 1992.

[138] van Beek P. and Dechter R. Constraint tightness versus global consistency. In *Proceedings of the 4th International Conference on Principles of Knowledge representation and Reasoning*, 1994.

[139] van Beek P. and Dechter R. On the minimality and global consistency of row convex constraint networks. In *Journal of the ACM*, 1995.

[140] van Hentenryck P. *Constraint satisfaction in logic programming.* The MIT press, 1989.

[141] van Hentenryck P., Deville P., and Teng C-M. A generic arc-consistency algorithm and its specializations. In *Artificial Intelligence 57*, 1992.

[142] Waltz D.L. Understanding line drawings of scenes with shadows. In P.W. Winston, editor, *The Psychology of Computer Vision.* McGraw-Hill, 1975.

[143] Ward A.C., Lozano-Pérez T., and Seering W.P. Extending the constraint propagation of intervals. In *AI EDAM, 4(1).* 1990.

[144] Yau M-M. and Srihari S.N. A hierarchical data structure for multidimensional digital images. In *Communications of the ACM, 26(7)*, 1983.